1987

Parallel and Vector Problems on the FLEX/32

H S. McFaddin

John R. Rice
Purdue University, jrr@cs.purdue.edu

Report Number:
87-661

PARALLEL AND VECTOR PROBLEMS ON THE FLEX/32

H. S. McFaddin
J. R. Rice

CSD-TR-661
March 1987

PARALLEL AND VECTOR PROBLEMS ON THE FLEX/32

H.S. McFaddin*
J.R. Rice**

Department of Computer Science
Purdue University
West Lafayette, Indiana 47907
CSD-TR-661

March 2, 1987

ABSTRACT

Earlier we proposed [Rice,1985] sixteen problems to test the effectiveness of languages in expressing parallel and vector computations. These problems were presented in ordinary notation (mathematics and English) plus four algorithmic forms: A) Fortran 77, B) Fortran 77 with extensions (resembling Fortran 8X), C) PROTRAN with extensions and D) Cyber 205 Fortran. We now present these problems programmed for the FLEX/32 multiprocessor in a Concurrent Fortran. Our objectives are twofold: 1) to show how these problems appear in this language (which is similar to those on several other multiprocessors), 2) to show the parallel efficiency achieved for these problems.

* This work was supported by IMSL, Inc.
** This work supported in part by ARO grant DAALO3-86-K-0106

1. INTRODUCTION AND SUMMARY

The purpose of this report is explore two properties of the FLEX/32 multiprocessor described in detail in [Flexible, 1985] (see Section 3 for a brief description). First, we wish to evaluate its effectiveness as a parallel computer (see [Houstis,et.al., 1987] for some earlier work). Second, we wish to evaluate the programming methodology required by its Concurrent Fortran Language. We have programmed sixteen problems on the FLEX/32 taken from [Rice, 1985] which are designed to test the effectiveness of programming languages for parallel and vector applications. These problems are also well suited to test the speedup obtained with parallel and vector computers.

The sixteen problems are summarized in the next section and the FLEX/32 and its Concurrent Fortran programming language are briefly described in Section 3. In Section 4 we present speedup curves for these problems using 2,3,4 and 5 processors of the FLEX/32. Each problem is parametrized so that both the amount of parallelism and the size of the computation can be increased. The data is in the form of efficiency

$$E = \frac{Time\ with\ one\ processor}{N\ *\ (Time\ with\ N\ processors)}$$

versus computer time with one processor. The range of times are from 200 to 2200 ticks (or 4 to 44 seconds), depending on the problem. A tick on the FLEX/32 clock is 1/50 second.

Appendix One contains the sixteen programs for the problems. These are complete programs just as used for the speedup evaluations. Each program starts with code for interactive input which is very similar from program to program.

The design of the FLEX/32 is best suited for one or a small number of applications that run for a long time. Such applications occur commonly in real-time control. The initiation of a set of parallel processes is a substantial activity: programs are loaded into the local memories of the processors, data are placed in common memories, tables are set up for parallel synchronization, etc. It is somewhat analogous to the link/load step in a sequential computation and takes at least one second of real time. The effect of this activity is magnified by the fact that some parts are sequential in the number of processors used.

This situation has a direct impact on synchronizations that must occur in some of the sixteen problems. The programs initiate the number of parallel processes to be used and then synchronization is carried out by traditional techniques (semaphores, critical variables, etc.) without using the parallel constructs of Concurrent Fortran.

An examination of the efficiency curves in Section 4 shows that they all start off quite low and one must have a 10 to 20 ticks sequential job for any of the programs to reach efficiencies of 80 or 90 percent. This is due to the start-up times for the parallel computations. Appendix Two contains the data on timing and efficiency.

A visual examination of the efficiency curves suggests that, with five processors, no program reaches 95% efficiency for a job less than 800-1000 ticks (about 15-20 seconds). We summarize that observations for the 14 problems (excluding numbers 8 and 16), the job sizes are measured in ticks on a single FLEX/32 processor.

	2 processors	5 processors
Job size to reach 75% efficiency		
Minimum	10	50
Typical	15-20	200-300
Maximum	100	3000
Job size to reach 95% efficiency		
Minimum	100	500
Typical	250	2000-5000?
Maximum	>2000	?

The reason that efficiencies of 95% are difficult to achieve is that most of the problems have some small "sequential" part (such as forming a sum) or have some synchronization code which is unneeded in a sequential computation.

Keep in mind that we have programmed most of these problems in the style that we think is typical of general programming: 1) some initial thought is given to the computations, 2) an approach is chosen, 3) a code is written and, 4) some testing for correctness is made. We did not invest the effort to obtain highly efficient codes for these problems.

Two problems, #8 (compute a divided difference table) and #16 (multiple linear equation solutions), have low efficiences for all job sizes. Our program for problem #8 is essentially sequential in behavior, we have not used an appropriate parallelization here. High parallel efficiency is only possible here when the problem size is very large compared to the number of processors. Our program for problem #16 also does not use an appropriate parallelization, a comparison with problem #10 shows this.

Writing efficient programs for the FLEX/32 requires one to become familiar with the details of the machine and to learn various synchronization techniques for parallel computations. The principal support that Concurrent Fortran provides for synchronization is shared critical variables, the explicit synchronization facilities are too inefficient to be used often. We made a few experiments of the effect of coarse grain verses small grain parallelism. For example, a computation with 3 processors with i going from 1 to 300 can be divided into 3 parts in two obvious ways:

$$1 \le i_1 \le 100,\ 101 \le i_2 \le 200,\quad 201 \le i_3 \le 300\ ,$$

or

$$i_1 = 1, 4, 7, \ldots,\ i_2 = 2, 5, 8, \ldots,\ i_3 = 3, 6, 9, \ldots,\quad .$$

If no further synchronization were required, then these two approaches give essentially equal efficiency. Of course, the coarse grain approach is more efficient if there is much synchronization required at all.

2. THE SIXTEEN PROBLEMS

Problem 1: Evaluate the trapezoidal rule estimate of an integral of $f(x)$:

$$T_N = h^*(\tfrac{1}{2}f(a) + \sum_{i=1}^{N-1} f(a+ih) + \tfrac{1}{2}f(b))$$

Problem 2: Compute the value of

$$e^* = \sum_{i=1}^{n}\prod_{j=1}^{m}(1. + e^{(-|i-j|)})$$

Problem 3: Compute the value of $S = \sum_{j=1}^{n}\prod_{i=1}^{m} a_{ij}$

Problem 4: Compute the value of $R = \sum_{\substack{i=1 \\ x_i \neq 0}}^{N} \frac{1}{x_i}$

Problem 5: One has a table of the i-th student's score on the u-th test. One is to

(a) list the top score for each student $= top_i$
(b) give the number of scores above the average $= NABOVE$
(c) increase all the above average scores by 10 percent
(d) give the lowest score that is above average $= BLOW\_ABOVE$
(e) say whether any student has all scores above average $= GENIUS$

Problem 6: Solve the tridiagonal system $Tx = y$ by the special, vector oriented algorithm of [Jordan, 1979]. The matrix T is represented by L,D and U, its lower diagonal, main diagonal and upper diagonal.

Problem 7: Compute polynomial interpolant values of $f(x)$ at five points using Lagrange interpolation formulas:

$$p(x) = \sum_{i=1}^{N} f(x_i) l_i(x) \qquad l_i(x) = \prod_{\substack{j=1 \\ i \neq j}}^{N} (x - x_j) / \prod_{\substack{j=1 \\ i \neq j}} (x_i - x_j)$$

Problem 8: The divided difference table for a set of data x_i, $y_i = f(x_i)$ is defined by the formulas

$$f[x_i] = y_i$$

$$f[x_i, x_{i+1}, \ldots, x_{i+k}] = \frac{f[x_{i+1}, \ldots, x_{i+k}] - f[x_i, \ldots, x_{i+k}]}{x_{i+k} - x_i}$$

The problem is to compute the first M columns of the divided difference table

$$D_{ik} = f[x_i, x_{i+1}, \ldots, x_{i+k-1}]$$

Problem 9: One has an array u_{ij} of values on an N by M grid and wants to replace each value by the average of its value plus those of all its neighbors. This is expressed by

$$u_{ij} = \left(\sum_{Neighbors} u_{ij} \right) / (Number\ of\ neighbors)$$

This computation is typical of what one does in solving partial differential equations, image processing and geometric modeling.

Problem 10: LU factorization of the N by N matrix $A = a_{ij}$ using Gauss elimination with pivoting.

Problem 11: Read sets of data d_i, $i = 1, \ldots, N$, trim the negative values to zero and large values to 1000, do a logarithmic transformation $d_i = log(1 + d_i)$ and compute the first four Fourier moments $\sum_{i=1}^{N} d_i \cos(\pi i/(N+1))$ then save these moments and the data ID in a data base.

Problem 12: Given the m by m matrix A, the 1 by m vector R, the m by 1 vector C and a number A, construct the array

$$ABIG = \begin{bmatrix} A & C \\ R & a \end{bmatrix}$$

Problem 13: For given vector **a, b, c, d** compute the new vector

$$a_i = a_i \sin b^i$$

$$\textit{If } a_i < \cos(c_i) \textit{ then } a_i = a_i + c_i$$

$$\textit{else } a_i = a_i - d_i$$

and compute

$$e = \sum_{j=1}^{ndim} a_i^2$$

Problem 13H: Modify Problem 13 for a machine that wants to have the computation split into 20 processes (e.g., such as the *HEP*).

Problem 14: Carry out a test of four methods to integrate three different functions with 10 different levels of accuracy each. Print out a table with all the results including the number of function evaluations and in each integration. This problem comes from [Rice, 1983], page 204.

Problem 15: Carry out a comparison of two types of interpolation points (equispaced and chebyshev spaced) for Hermite interpolation using piece-wise cubic polynomials. The interpolant's value v at y can be expressed as

$$v(y) = \sum_{j=1}^{N} f(x_j)h_{1j}(y) + f'(x_j)h_{2j}(y)$$

where $h_{1j}(x)$ and $h_{2j}(x)$ are suitable basis functions that depend on the N interpolation points x_j. This problem comes from [Rice, 1983], pages 93, 98 and 380-381.

Problem 16: Solve a matrix equation $Ax = B$ where A is an N by N Hilbert matrix and B is an N by 4 matrix. The matrix order N takes on the values 4, 8, 12, 16 and 20 and the B column-vectors are, respectively, the first column of the identity matrix, all 1's, a 0.01 random pertubation of all 1's, and alternating +1, -1.

3. THE FLEX/32 MULTICOMPUTER AND ITS CONCURRENT FORTRAN

3.1. FLEX/32 Architecture

The FLEX/32 is a MIMD (Multiple Instruction Stream Multiple Data Stream) computing device which may be configured to operate up to 20 independent processing units. Each processor may access a shared common memory via a common bus, as well as its own local memory. The programs included in this report primarily use the shared memory – local memories are used only to store intermediate computations. An effort is made to reduce shared memory accesses, although the report [Houstis et.al., 1987] indicate that the effect of memory contention is negligible. Figure 1 shows a block diagram of the structure of the FLEX/32, see [Flex,1985] for more details.

The FLEX/32 operated by the Computer Science Department is currently configured with 7 processors. One processor runs multi-user UNIX for program development and has 4 Mbytes of local memory. The remaining six operate in batch-parallel mode under the MMOS operating system, each has 1 Mbyte of local memory. There are six shared memory modules with 512 Kbytes of memory each.

3.2. FLEX/32 Concurrent Fortran

3.2.1 *Implementing Parallelism on the FLEX/32*

FLEX/32 Concurrent Fortran is an example of a parallel language constructed on top of an older one. The new features range in functionality from high level (block structures, conditional waits, etc.) to low level (explicit process forks).

A parallel program in Concurrent Fortran may be viewed as a main procedure which has been downloaded from the UNIX development computer onto one of the MMOS computers. Eventually, the main program may fork into a collection of independent processes executing concurrently on one or more of the MMOS computers. The machine is then operating in parallel as illustrated in Figure 2.

The forking operation has been observed to be expensive, requiring as much as one second of real time. Thus, it is a bad idea to write programs which repeatedly fork and join (i.e., return to MAIN) as a form of interprocess synchronization. For example, consider a program which performs Gaussian Elimination on a matrix in shared memory. We might choose to compute each submatrix in parallel, and thus approach the problem as shown in Figure 3. This is a bad approach. Instead, the programmer should use a single fork (or a constant number of them) per program run, and relegate synchronization duties to the processes spawned. In the programs of this report, for example, synchronization is implemented by incrementing and inspecting integer variables (semaphores) in the shared memory.

From the viewpoint of the programmer, then, parallelism in Concurrent Fortran should be coarse grained – the chunks of code parallelized are on the order of entire programs, instead of a few lines. If parallelization is synchronized and done at the "few lines of code" level, then it

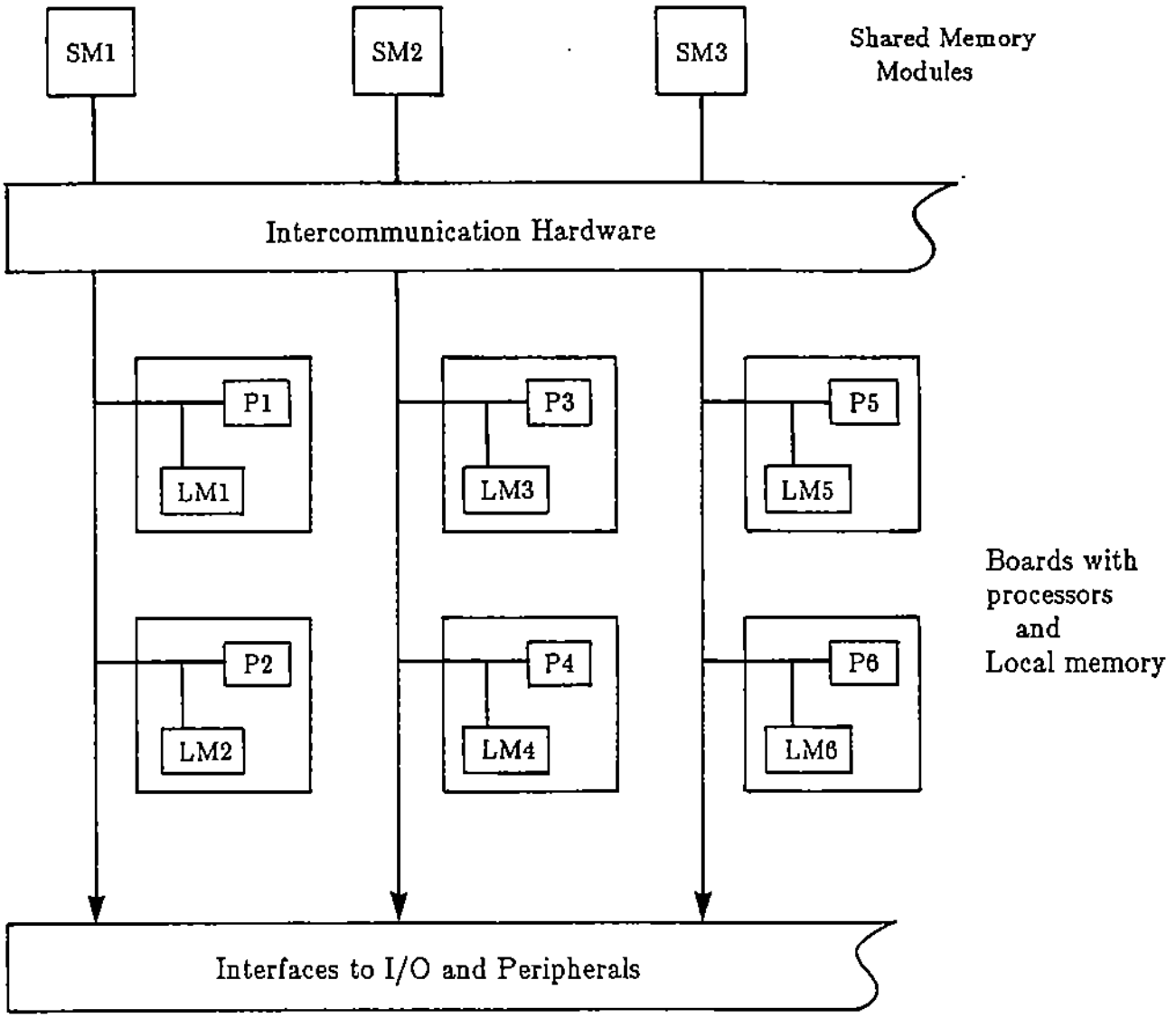


Figure 1. Schematic of the FLEX/32 architecture. There may be up to 10 shared memory modules (SM1, SM2, ...,) and up to 20 processor boards with a processor (P1, P2, ...,) and local memory (LM1, LM2, ...,).

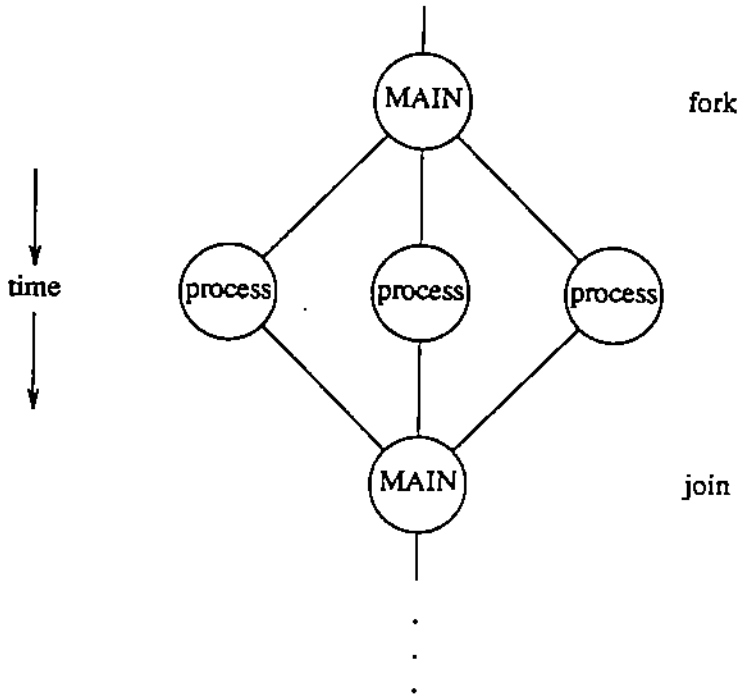


Figure 2. Schematic of the process fork and join of Concurrent Fortran.

must be synchronized using shared variables which adds a significant but not overwhelming additional computation. This reflects the design intentions of the FLEX/32, as a general purpose industrial machine capable of concurrently operating several dissimilar processes over long periods of time.

3.2.2 *Declaring Shared Memory*

Shared memory must be made visible to every process wishing to access and/or change it. This is done by declaring shared variables. The syntax is to precede a normal FORTRAN declaration by the word **shared**.

Example:

```
shared real / r1 /  a(100,100) pivot
shared real / i1 /  isynch, ipvt, row(100)
```

Shared blocks should be named (if there is more than one) and shared variable declarations may not involve parameter values.

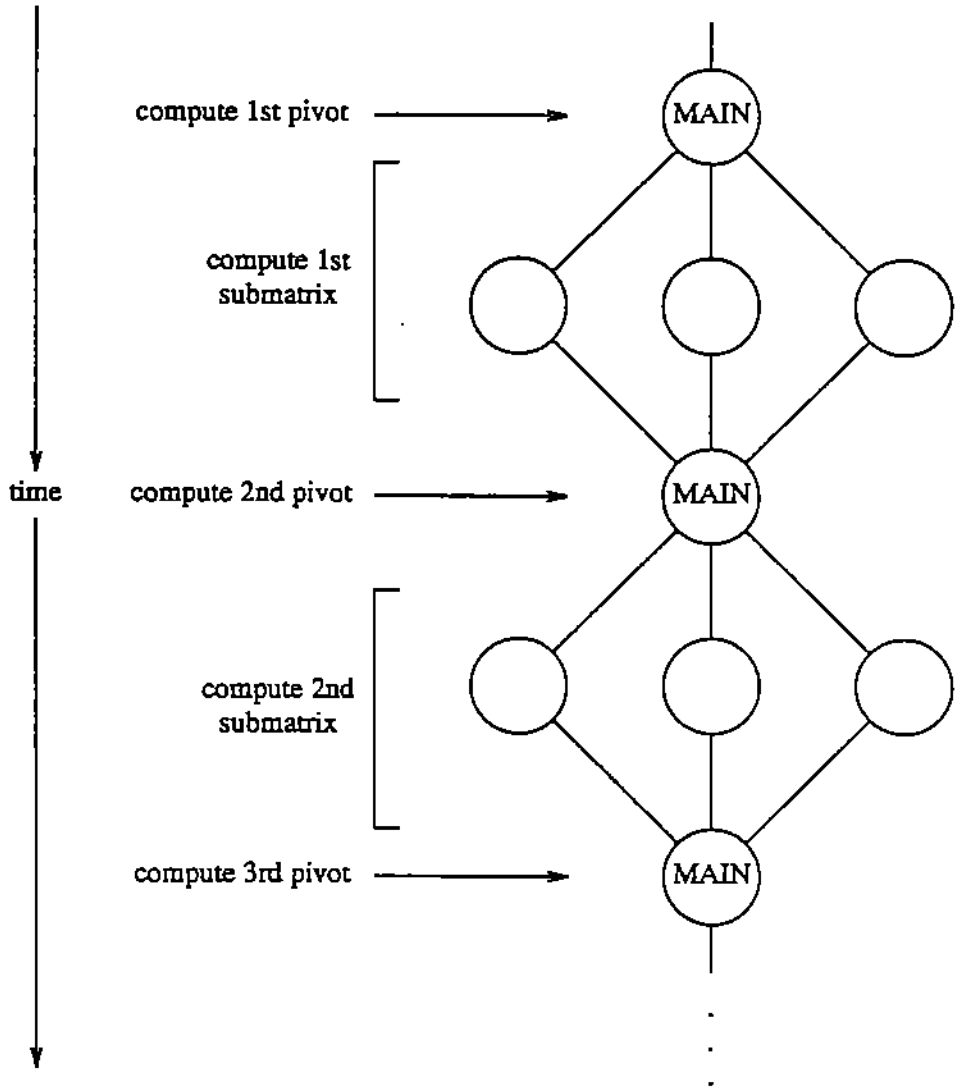


Figure 3. Schmatic of an inefficient approach to Gauss Elimination because of excessive forking which is very expensive in the FLEX/32.

3.2.3 *Creating Processes*

To create a new process, a parent process, such as MAIN, must supply an integer into which the process id of the child is to be written, provide code to begin executing (in the form of a subroutine call) and specify the computer on which to operate. The syntax of the creation statement is

process (proc_id, subroutine_name, computer_number)

The code to be mapped to the specified computer is determined by mimicking the given subroutine call – addresses (l-values of argument expressions) are placed in stacks, etc. However, the

spawned process does not begin execution immediately. Rather, the parent process is free to execute other code, possibly spawning other processes, before initiating actual execution.

Example:

```
      do 50 i = 3,6
         arg(i) = i
         process (pid(i), code(arg(i)), i)
50    continue
         .
         .
         .

--now start above processes--
```

Here, the programmer must be careful. Since addresses (call by reference) are passed to subroutines in Fortran, the programmer must bear in mind that the values at those addresses are determined at the time execution begins, but the location references are established at the time of the process statement. To force the evaluation of arguments at the time of the process statement, an intermediate structure should be used, such as the array "arg" above. Had we specified code(i) instead of code(arg(i)) as the subroutine call, then the process would access the value in the integer i, not at the time of the process statement, but later, at the time process execution is initiated. This would give $i = 6$ for all processes.

The process construct interferes with programmer abstraction on two counts. First, the programmer must specify explicitly which computer is to execute the new process. This requires knowledge of the current configuration of the machine. Secondly, the programmer must chronologically separate the determination of the address of a subroutine argument (the l-value of the argument expression), occurring at the time of the process statement, and the actual value (r-value) of the argument, determined later, at process initiation. The child process cannot be viewed as a subroutine call at the point of the process statement, for the r-value is not determined. Neither can it be viewed as a sub-routine call at the point of process initiation, for the l-value was determined earlier.

3.2.4 *Block Structures for Process Generation*

As explained above, the programmer may choose to defer initiation of one or more spawned processes, beginning them all simultaneously. This is accomplished by enclosing the process statements in a COBEGIN block:

```
COBEGIN
--process statements and other code--
END COBEGIN
```

Initiation of all processes within the block is deferred until the bottom of the block is reached. The parent process may continue executing at that point.

Another alternative is to have spawned process begin executing immediately, but force the parent process to wait until all children have returned before proceeding. This construct is the COEND block:

```
COEND
-- process statements and other code --
END COEND
```

The COBLOCK construct combines the above two, deferring initiation of child processes to the bottom of the block, and forcing the parent process to wait until all child processes in the block have terminated

```
COBLCOK
-- process statements and other code --
END COBLOCK
```

3.2.5 *Conditional Waiting*

Concurrent Fortran provides a facility to have a process wait until a condition is satisfied:

```
WHEN(cond)statements
```

or

```
WHEN(cond)statements
-block of statements -
END WHEN
```

This is spin waiting.

3.2.6 *Critical Access*

When several processes wish to update a shared memory location, it is often necessary that only one process have access at a time. Suppose, for example, several processes P_i have each generated a local value X_i and we wish to form the sum $S = \sum_i X_i$. Then while some process P_i executes $S = S + X$ no other process should read S. Process P_i must have exclusive access to S. Concurrent Fortran provides two facilities to ensure exclusive access to shared memory constructs.

The first is via the WHEN structure above. All shared variables in the conditional part of a WHEN statement are locked while the conditional is being evaluated and, if successful, while the code within the WHEN block is executing. Thus, it is possible to *implicitly* generate a block of code having exclusive access to portions of shared memory. Notice in this example that process P_i does not really want to wait on a condition -- it simply wants exclusive access to S while it executes the (short) block of code

```
S = S + X
```

We must use a conditional which always evaluates to .TRUE. and contains the variable S. The typical technique, then, is to use a block such as

```
WHEN ( S .EQ. S ) THEN
  S = S + X
END WHEN
```

The WHEN block implements exclusive access by calling the MMOS system routines CFlock and CFulck, which lock and unlock, respectively, shared variables. The programmer may wish to explicitly perform locking by calling the routines himself:

call CFlock(ICFret, # of variable names, list of variable names)
and
call CFulck(ICFret, # of variable names, list of variable names)

(ICFret is an integer through which error codes are returned) In our example, we would use the code

```
call CFlock( ICFret, 1, 'S' )
  S = S + X
call CFulck( ICFret, 1, 'S' )
```

We make two observations regarding this locking scheme. First, the programmer must bear in mind that any shared variables mentioned in the conditional of a WHEN statement are locked for the duration of the block, not just for the conditional evaluation. Thus, the WHEN block should be as small as possible, so that the process does not tie up the shared variable any longer than necessary. In fact, we suggest that the block form of the WHEN structure be used only when the programmer desires BOTH the "wait" functionality and the exclusive access functionality of the WHEN block. Otherwise, the simpler form of the WHEN

WHEN(condition) continue

should be used when only waiting is required, and explicit lock calls when only exclusive access is required. Thus, we would classify the first locking scheme given above as bad.

The second observation is that the locking routines tend to lock "too much". Since the locking routine uses the variable name as a key, the programmer is not allowed to lock only *portions* of arrays. Here, caution is advised, for the programmer could inadvertantly create a deadlock situation by assuming only part of an array was locked. This is especially likely if the programmer is in the habit of using WHEN structures to *implicitly* lock arrays.

3.2.7 *Synchronization*

Concurrent Fortran provides STATIC communication channels which allow one process to trigger exceptions in another process. In the programs of this report, we have chosen a more dynamic, and higher level, approach to synchronization, by implementing semaphores in shared memory.

Suppose we have a section of code C which we wish to be executed simultaneously on each of N processors and we wish all processors to halt together before continuing. An example of such a code fragment would be a single submatrix reduction in Gaussian elimination – no processor should proceed to the next submatrix reduction until all have finished the current one. Assume there is an integer variable ISYNCH in shared memory (and thus visible to all N processes) which had value S before any of the processes began executing code C. To synchronize the processes we require that each increment the semaphore ISYNCH and then wait until the other processors have done so. A typical code fragment would be

```
      .
      .
      C (the code fragment being synchronized)
      .
      .
call.CFlock(1CFret,1,'isynch')
   ISYNCH = ISYNCH + 1
call CFulck(1CFret,1,'isynch')
WHEN (ISYNCH .GE. S + N) CONTINUE
```

The system calls provide the locking functionality needed to correctly increment the semaphore, while the WHEN statement provides the waiting functionality.

4. THE PARALLEL EFFICIENCY CURVES

We plot efficiency E versus the job size (ticks for a sequential computation) for each of the sixteen problems. Four curves are given for each problem corresponding to using 2, 3, 4 or 5 processors. As one would expect, the efficiency is generally decreasing as one increases the number of processors. The timings are made using the standard FLEX/32 mechanism (one counts "ticks" which are 20 milliseconds each). It has been found to be reliable (repeatable) and consistent with other timing data for these NS32032 processors. Data from which these curves are derived is given in Appendix Two.

5. REFERENCES

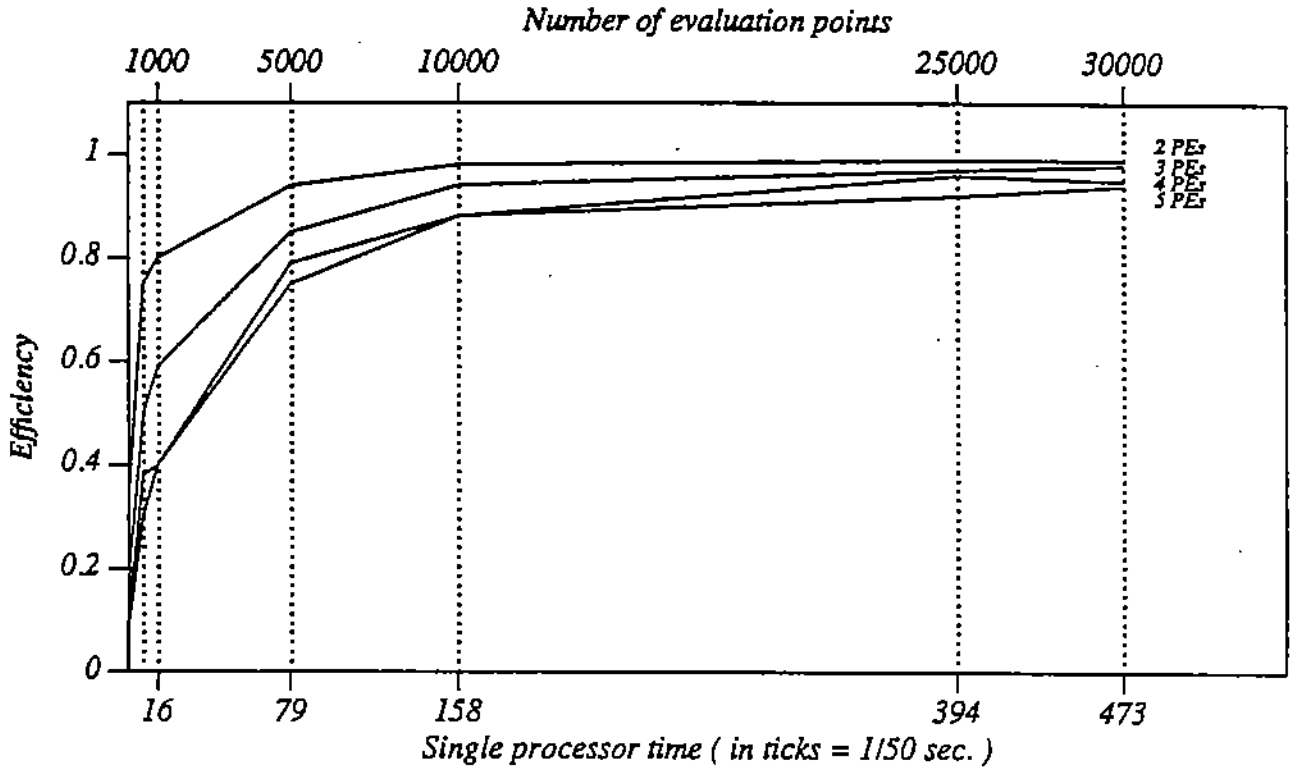
Flexible Computer Corporation, *FLEX/32 Multicomputer System Overview*, Doc. No. 030-0000-002, (June, 1985).

C.E. Houstis, E.N. Houstis and J.R. Rice, Partitioning PDE Computations: Methods and performance evaluation, *Parallel Computing* (1987).

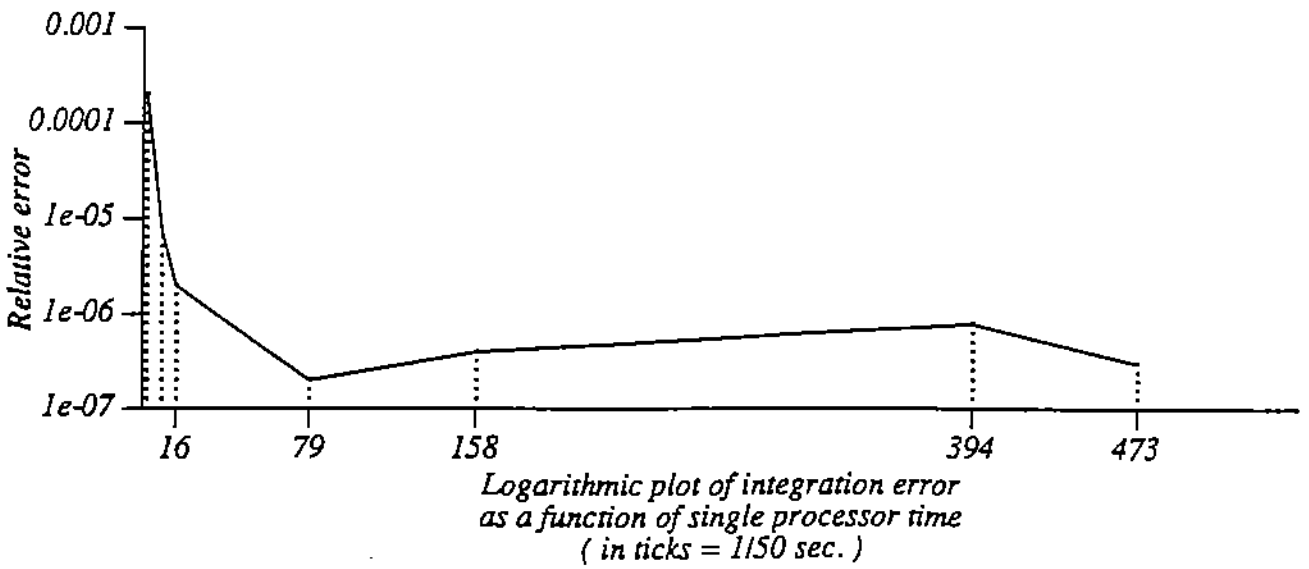
J.R. Rice, *Numerical Methods, Software and Analysis*, McGraw Hill, 1983.

J.R. Rice, Problems to test parallel and vector languages, CSD-TR 516, Department of Computer Science, Purdue University, (May, 1985), 95 pages.

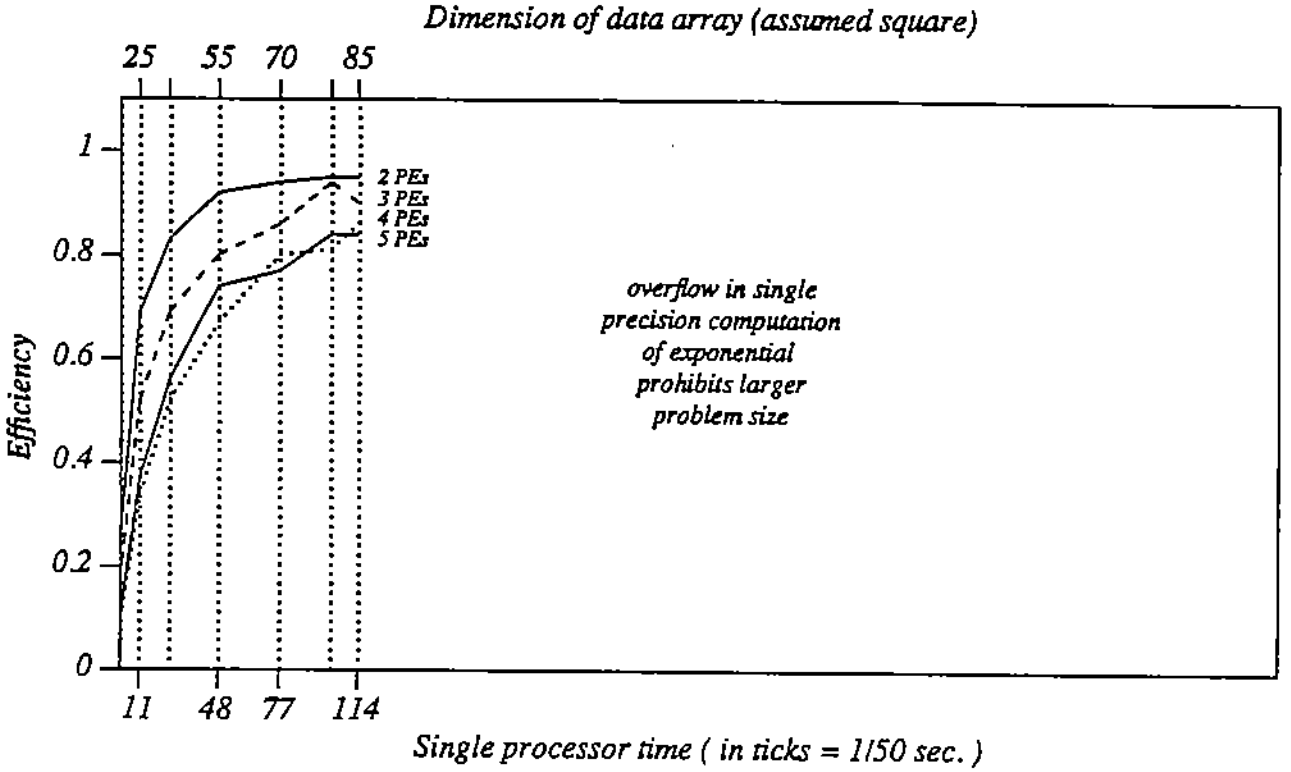
Problem 1
Integration by Trapezoidal rule



$$Efficiency = \frac{(Single\ processor\ time)}{N \times (N\ processor\ time)}$$

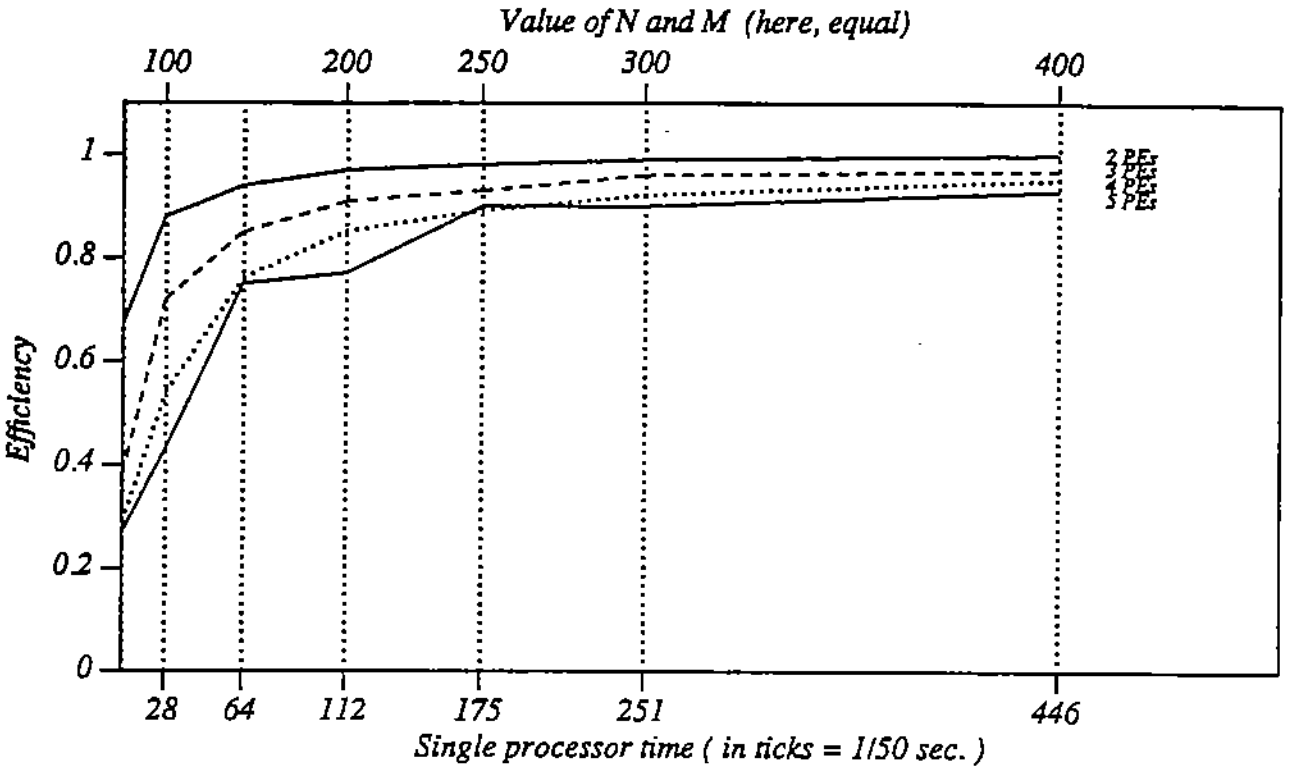


Problem 2
E-STAR



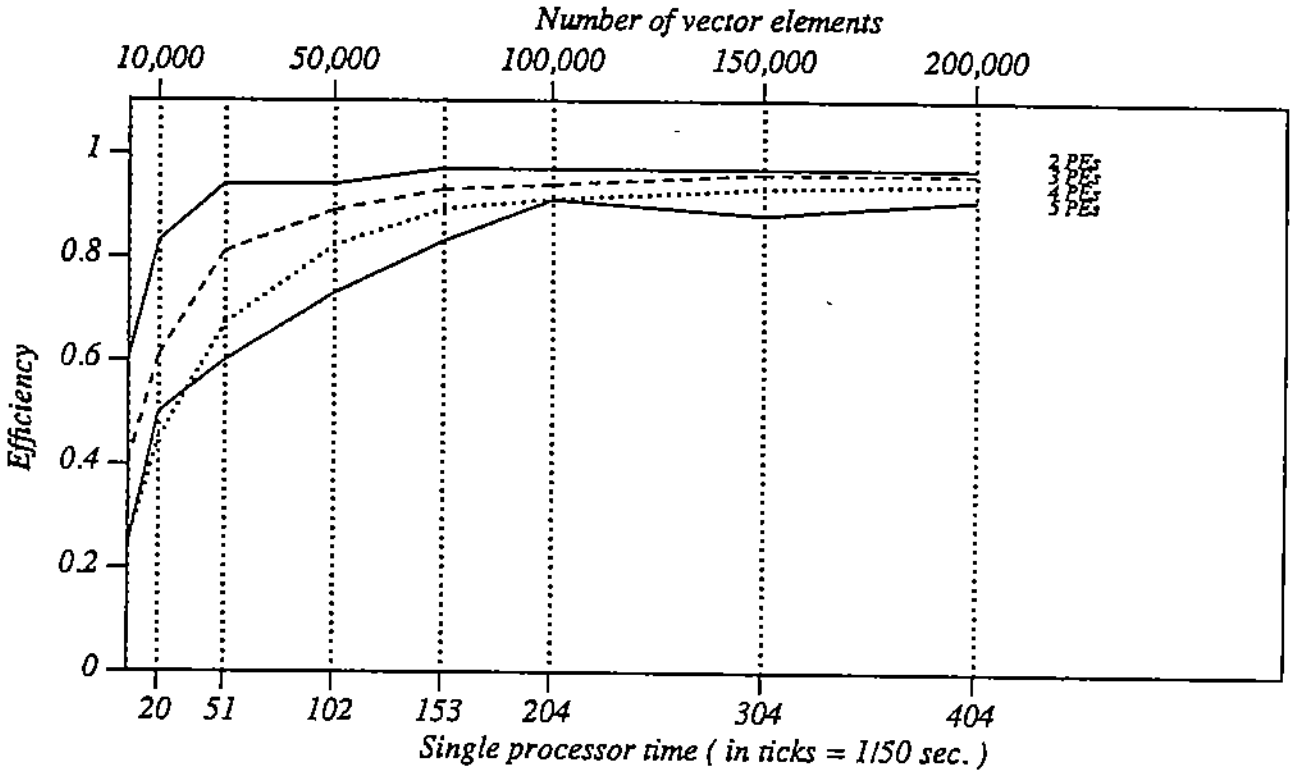
$$Efficiency = \frac{(Single\ processor\ time)}{N \times (N\ processor\ time)}$$

Problem 3
Generalized E-STAR



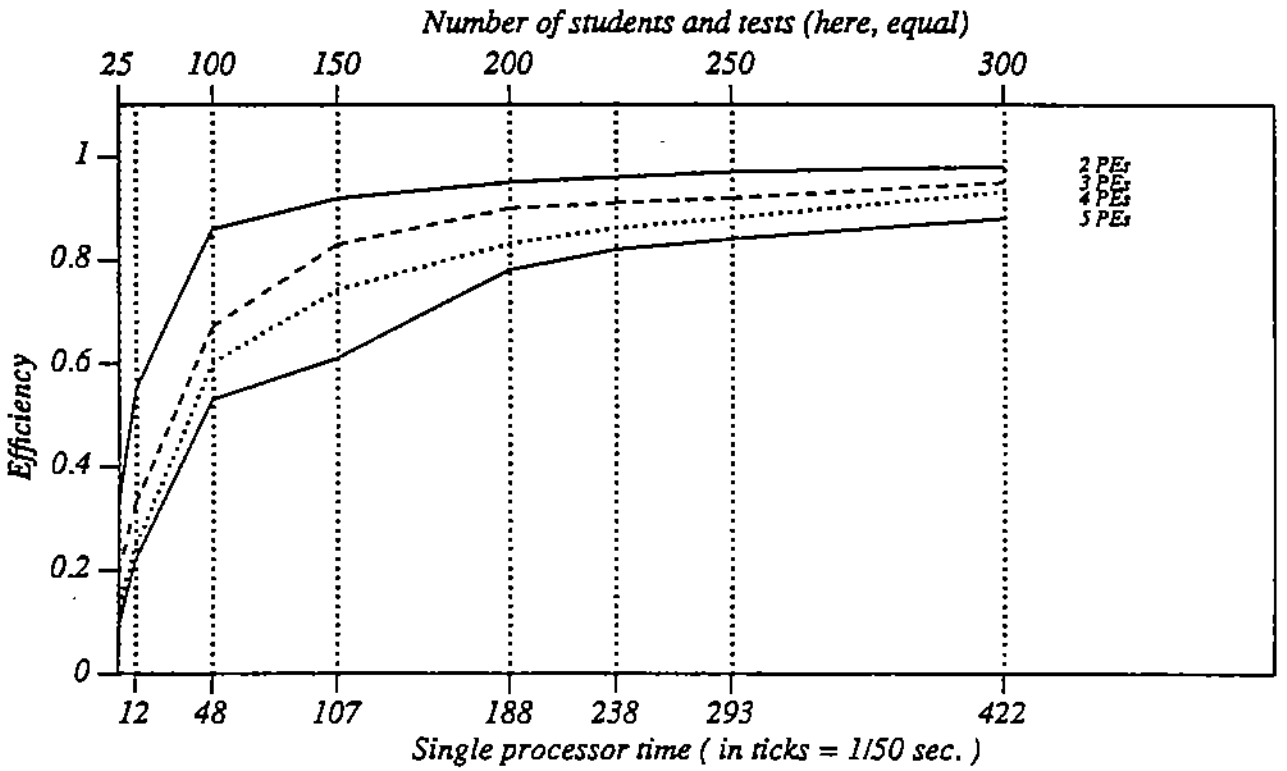
$$Efficiency = \frac{(Single\ processor\ time)}{N \times (N\ processor\ time)}$$

Problem 4
Sum of inverses



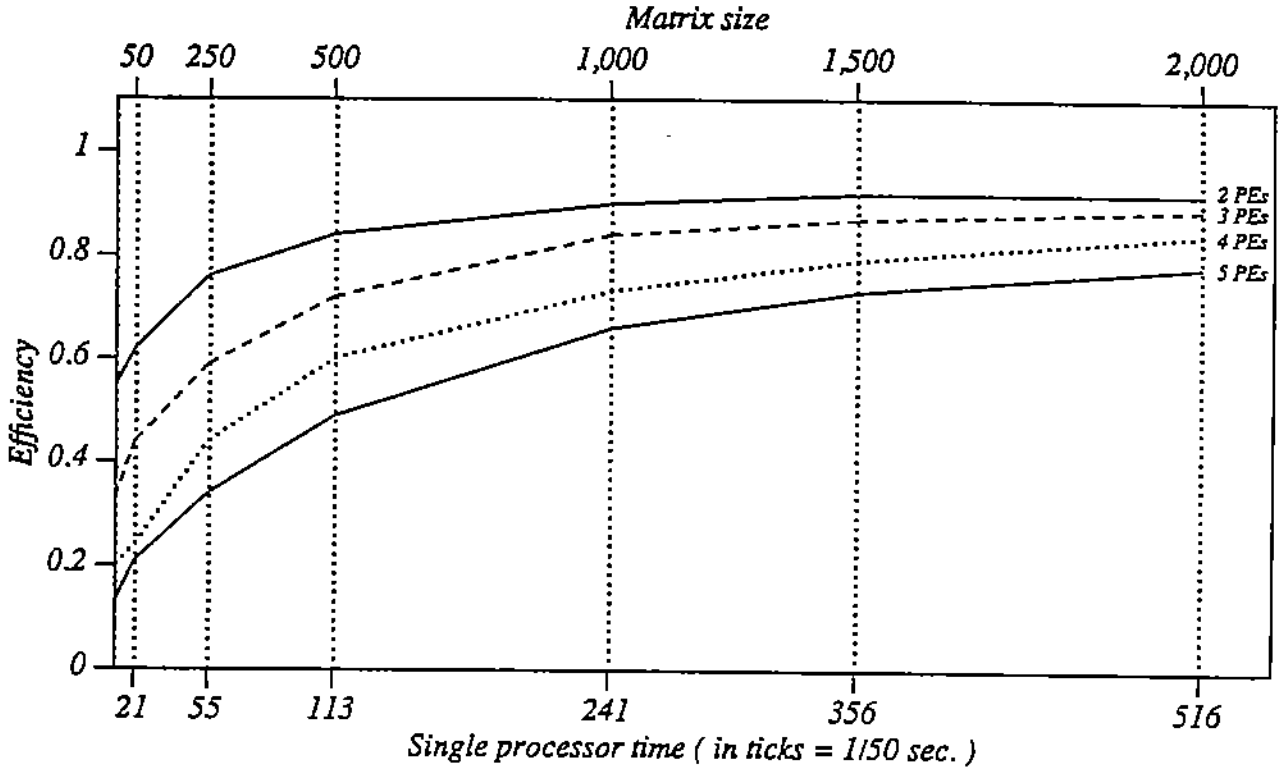
$$Efficiency = \frac{(Single\ processor\ time)}{N \times (N\ processor\ time)}$$

Problem 5
Grader program



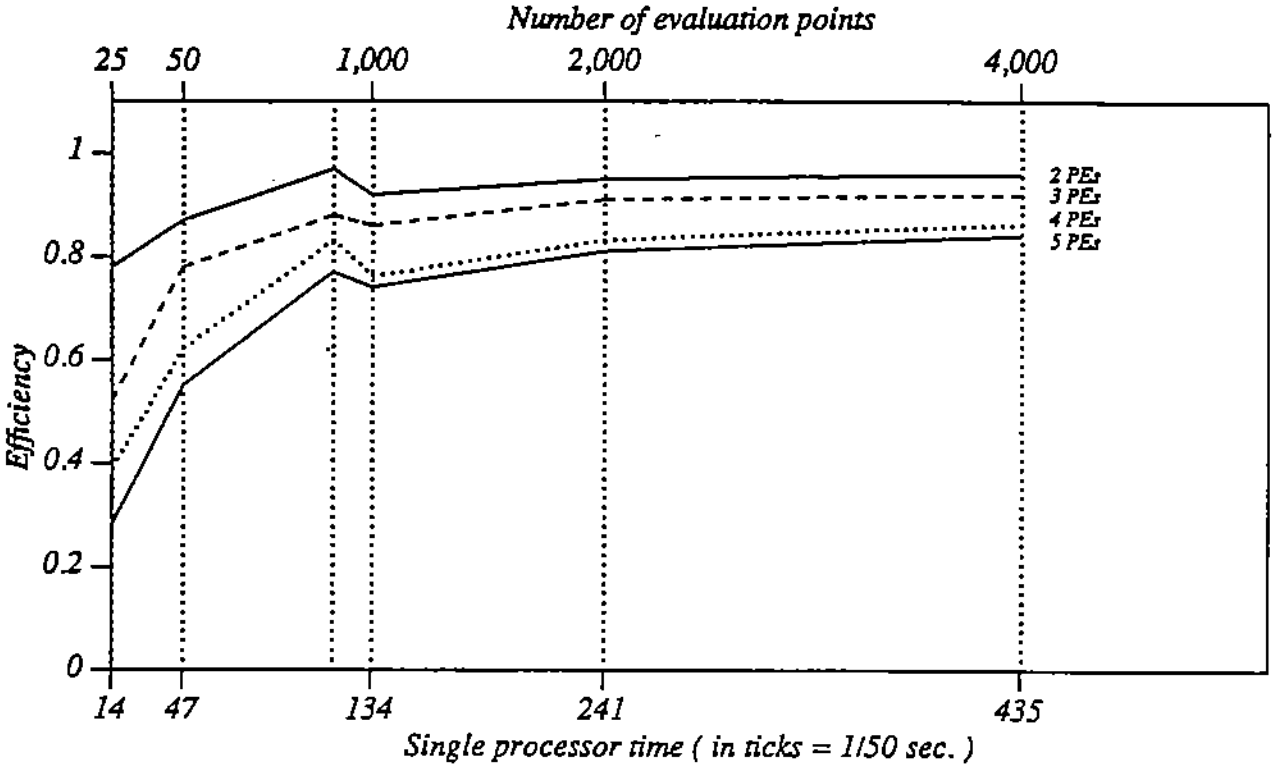
$$Efficiency = \frac{(Single\ processor\ time)}{N \times (N\ processor\ time)}$$

Problem 6
Tridiagonal solver



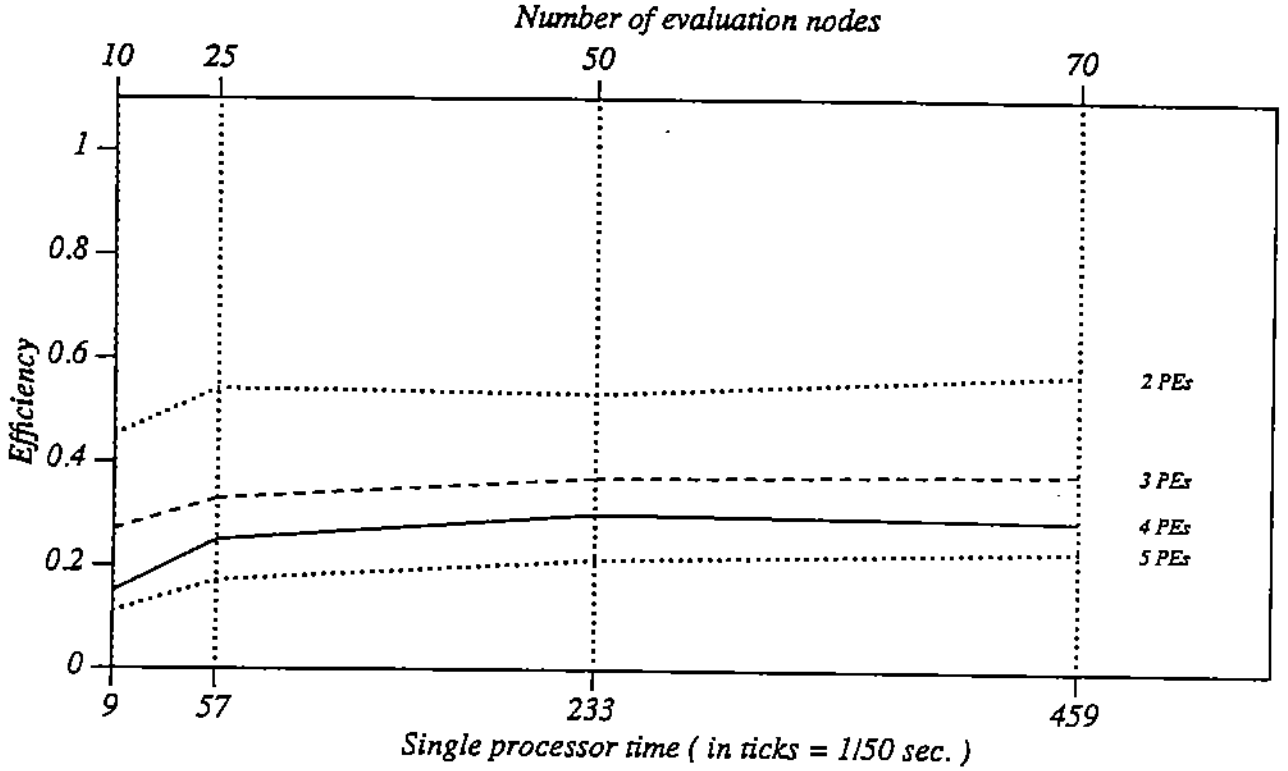
$$Efficiency = \frac{(Single\ processor\ time)}{N \times (N\ processor\ time)}$$

Problem 7
Polynomial interpolation



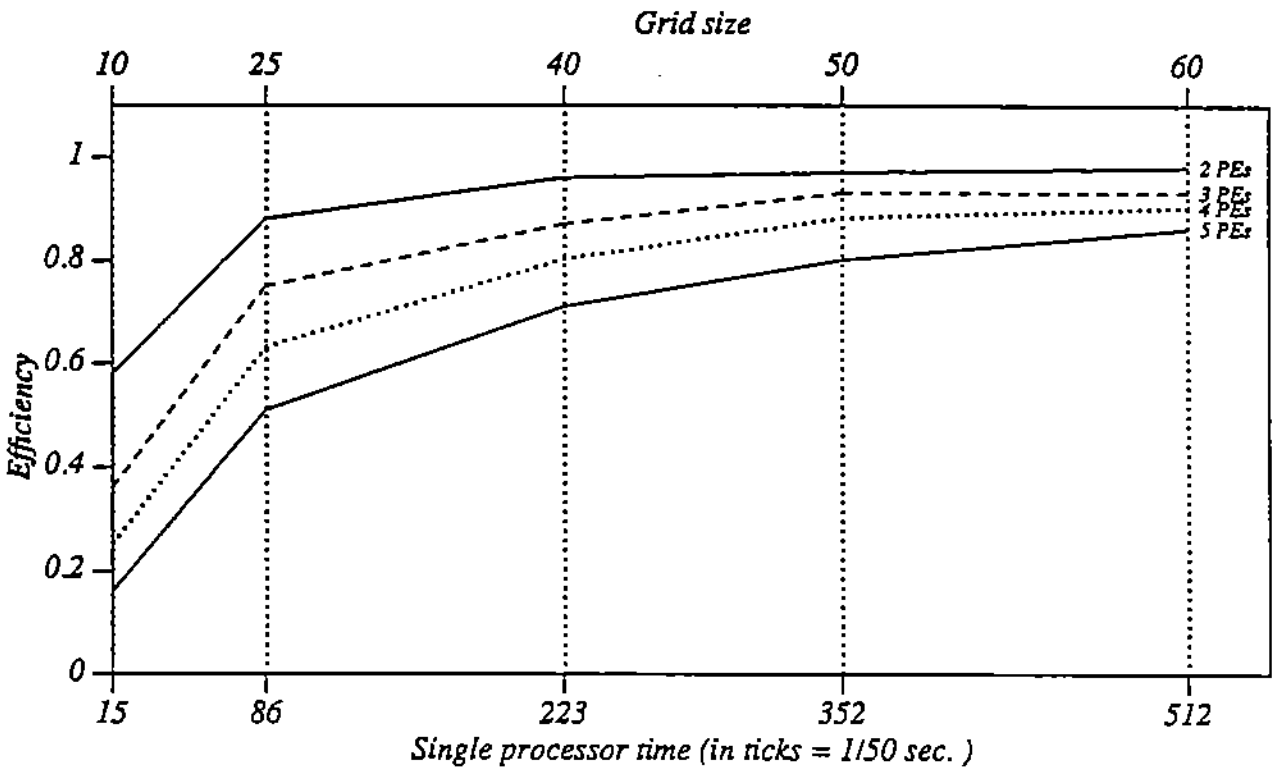
$$Efficiency = \frac{(Single\ processor\ time)}{N \times (N\ processor\ time)}$$

Problem 8
Divided Difference Table



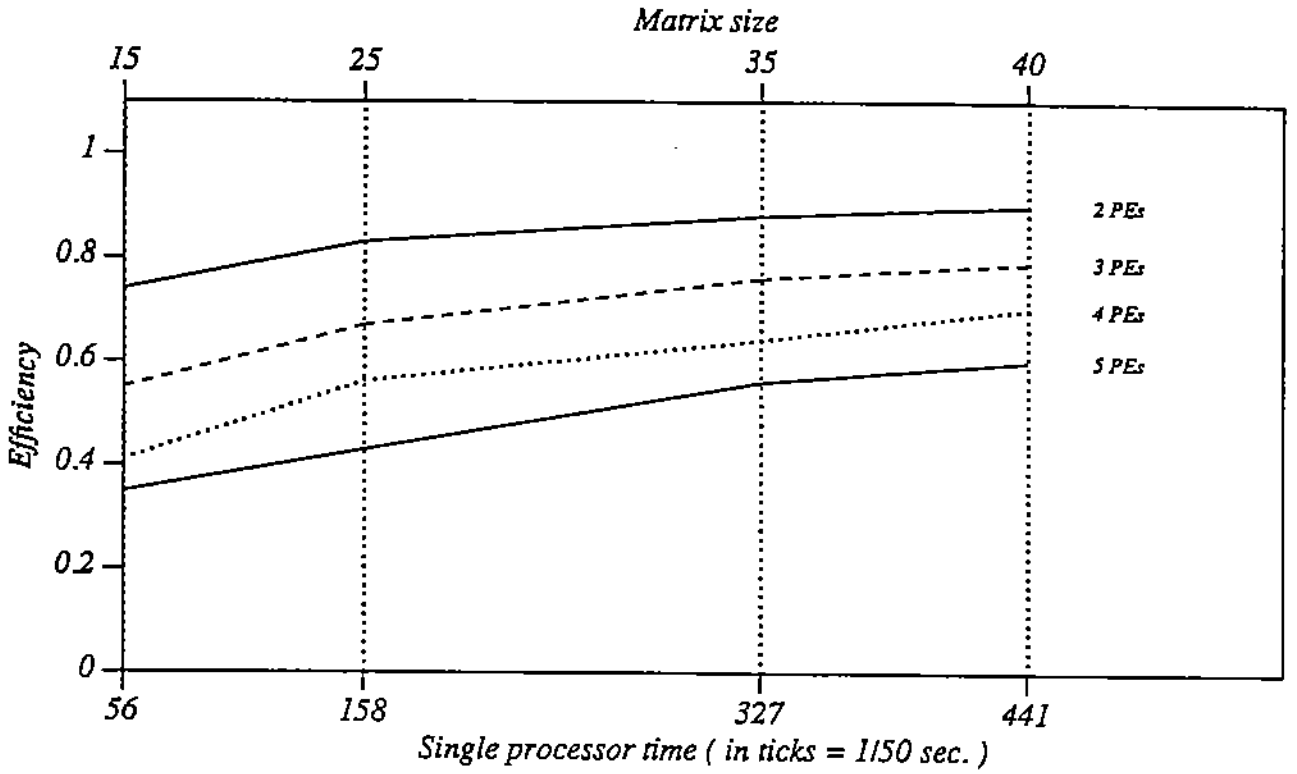
$$Efficiency = \frac{(Single\ processor\ time)}{N \times (N\ processor\ time)}$$

Problem 9
Image Refinement
10 passes



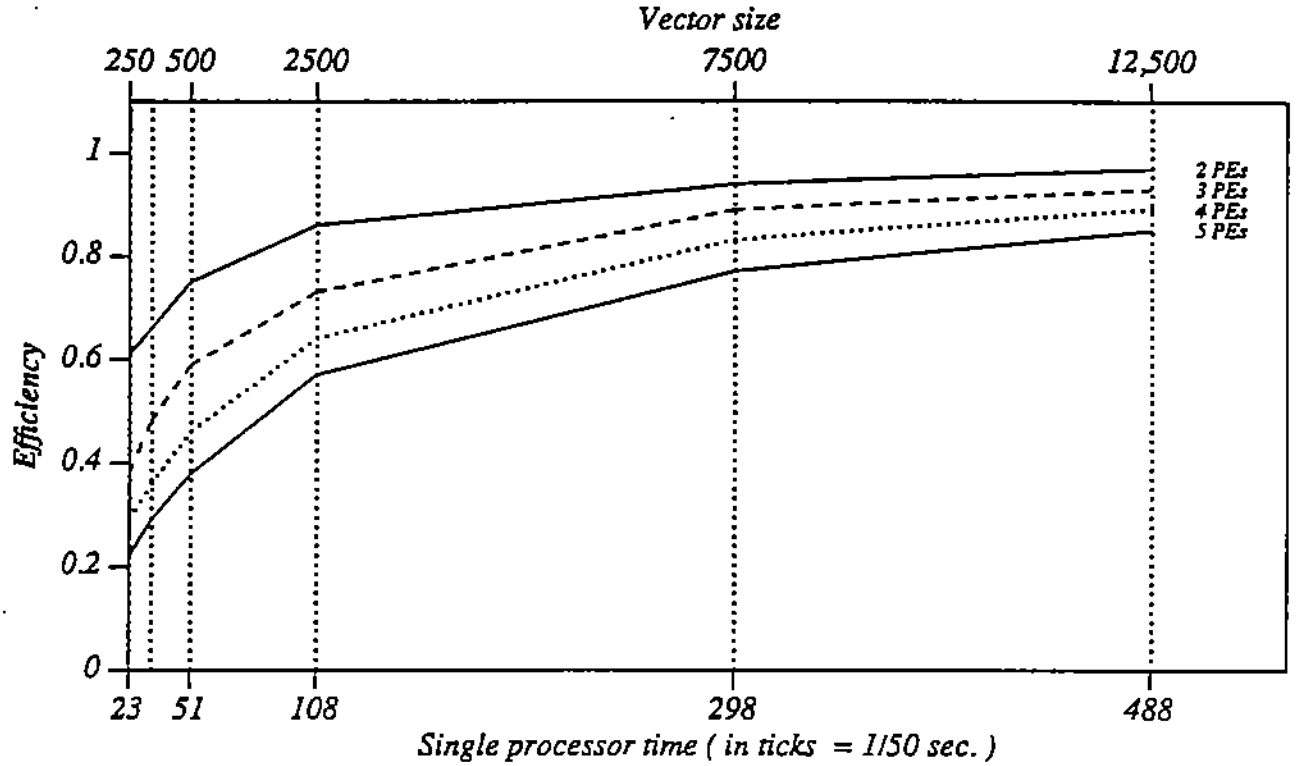
$$Efficiency = \frac{(Single\ processor\ time)}{N \times (N\ processor\ time)}$$

Problem 10
Gaussian Elimination



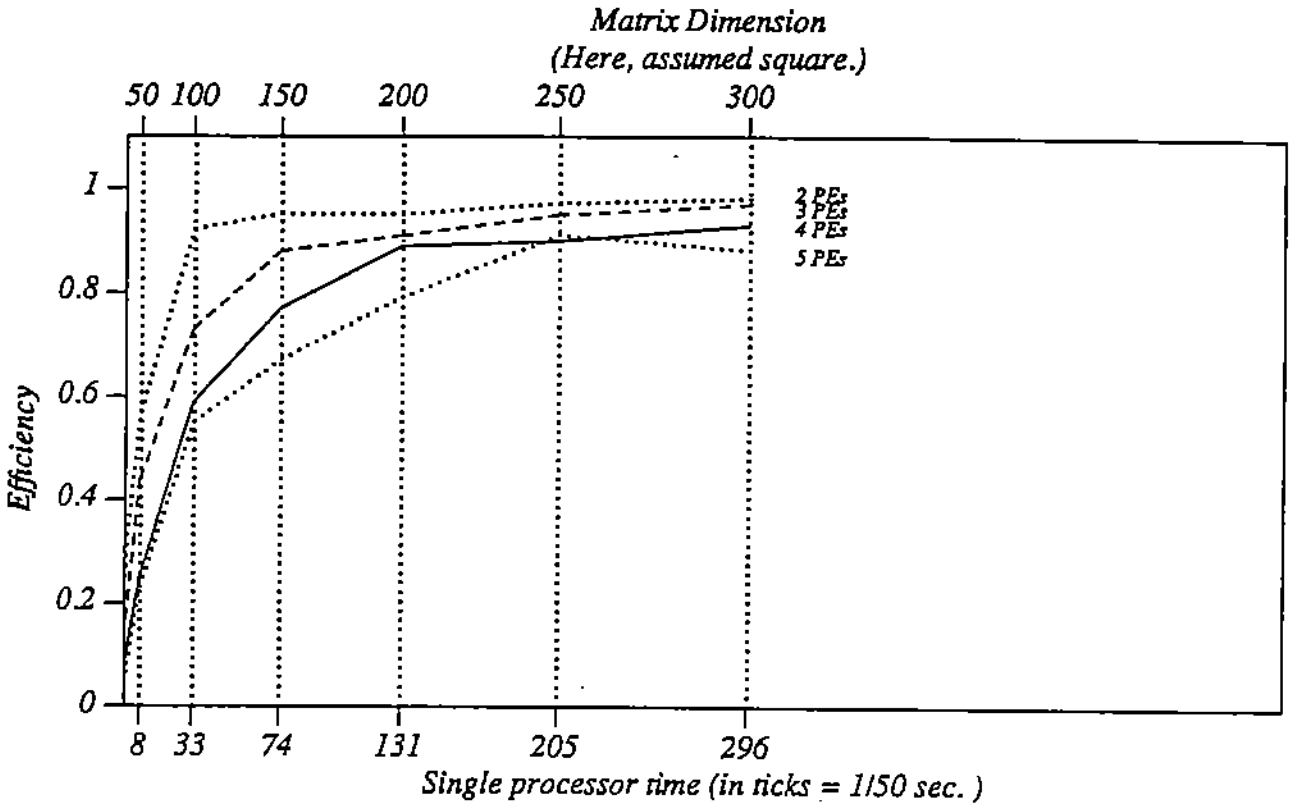
$$Efficiency = \frac{(Single\ processor\ time)}{N \times (N\ processor\ time)}$$

Problem 11
Data Filtering



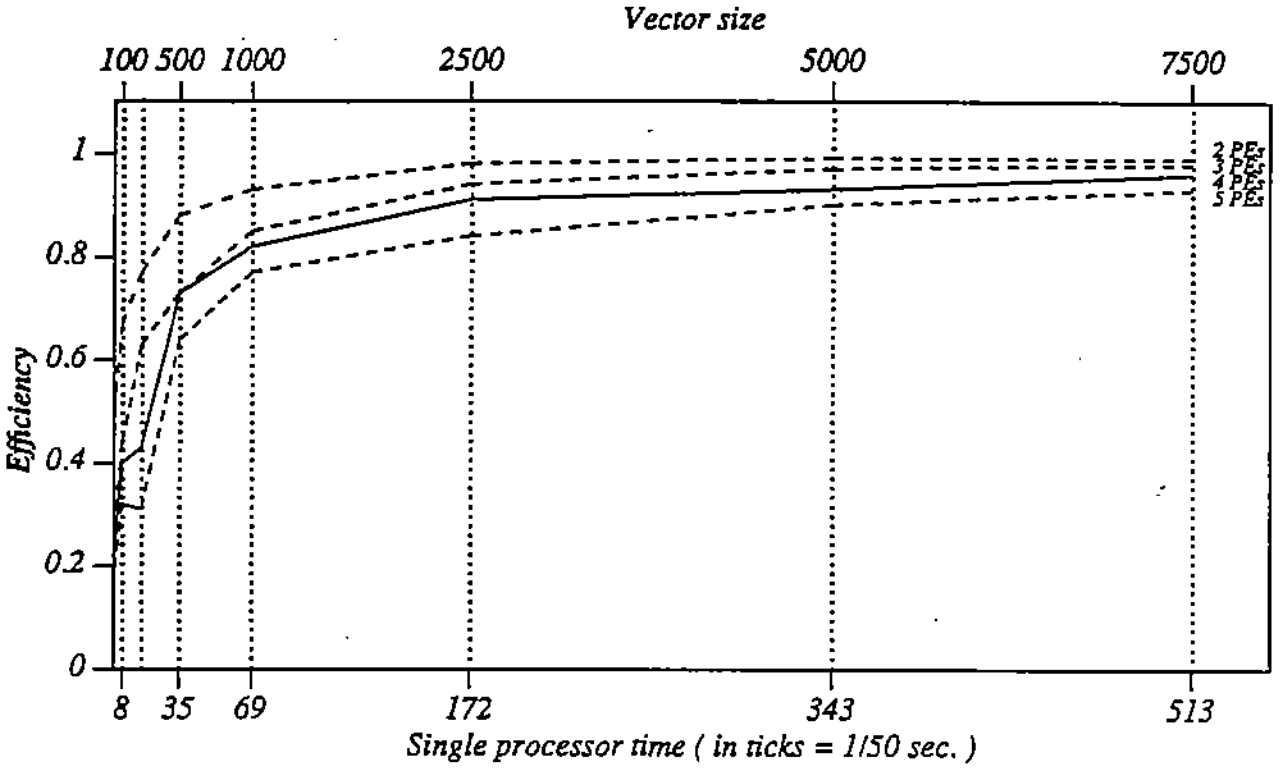
$$Efficiency = \frac{(Single\ processor\ time)}{N \times (N\ processor\ time)}$$

Problem 12
Data Movement



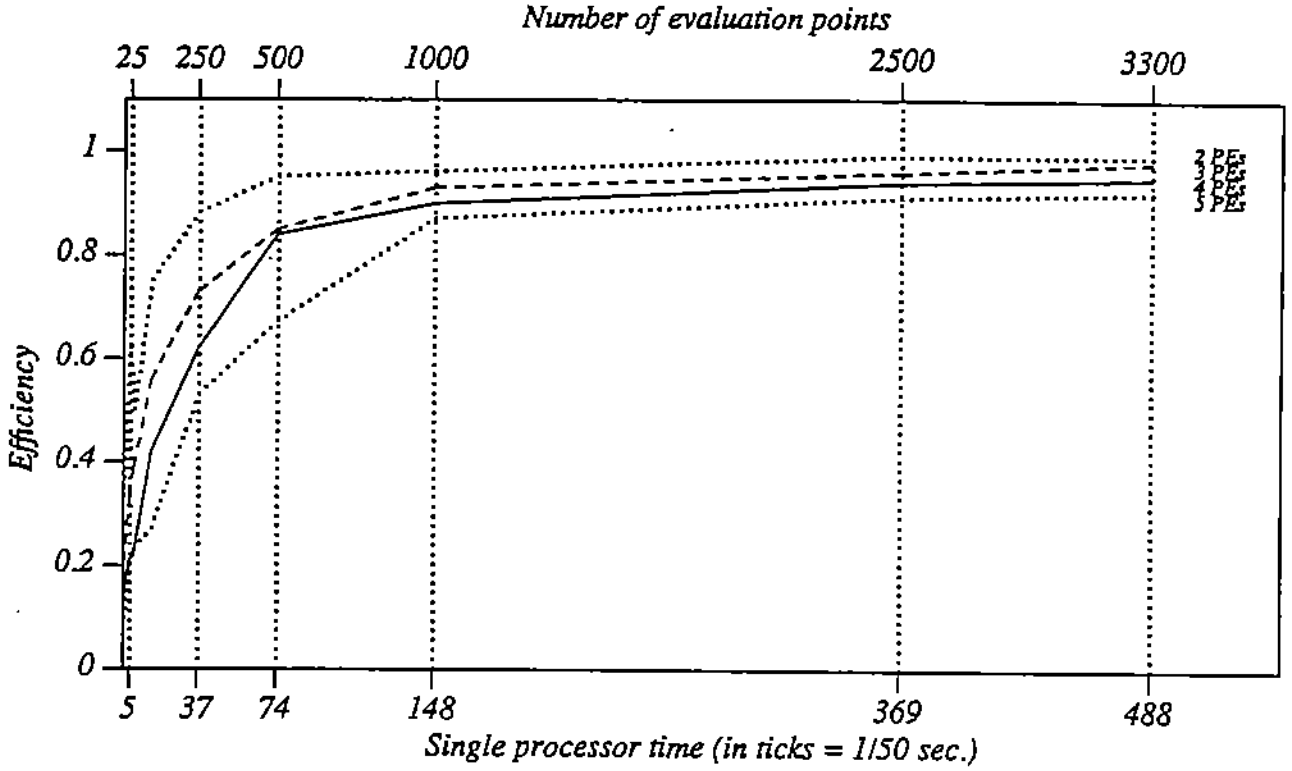
$$Efficiency = \frac{(Single\ processor\ time)}{N \times (N\ processor\ time)}$$

Problem 13
Vector Manipulation



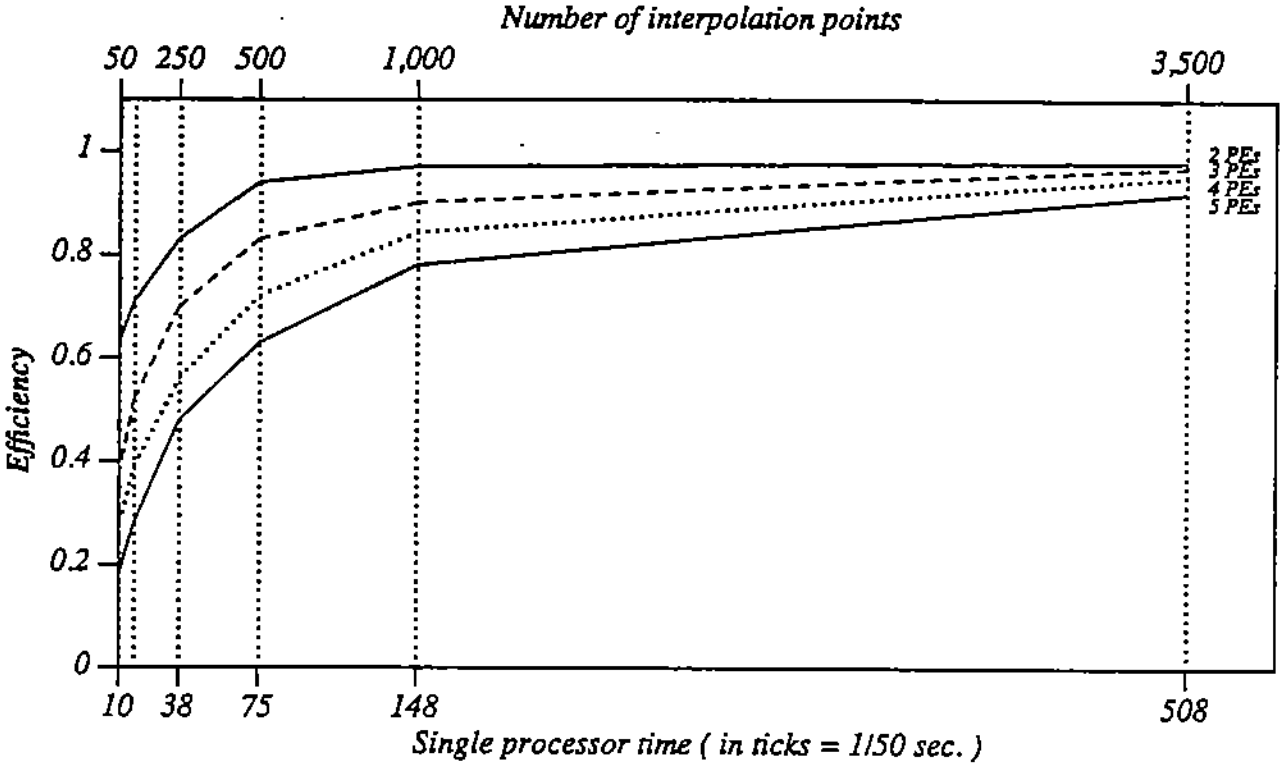
$$Efficiency = \frac{(Single\ processor\ time)}{N \times (N\ processor\ time)}$$

Problem 14
Integration tests



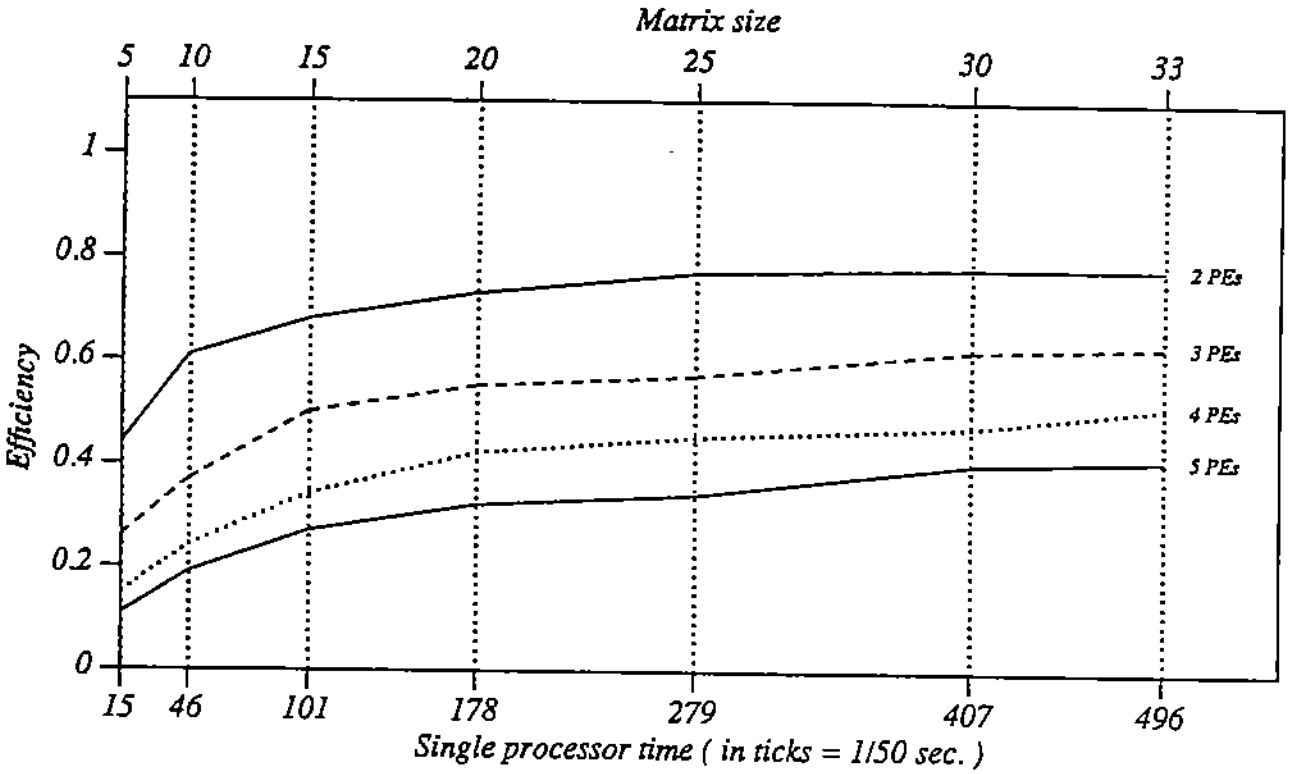
$$Efficiency = \frac{(Single\ processor\ time)}{N \times (N\ processor\ time)}$$

Problem 15
Interpolation



$$Efficiency = \frac{(Single\ processor\ time)}{N \times (N\ processor\ time)}$$

Problem 16
Hilbert Matrix System



$$Efficiency = \frac{(Single\ processor\ time)}{N \times (N\ processor\ time)}$$

APPENDIX ONE: THE FLEX/32 PROGRAMS

```
c       Problem 1
c       Trapezoidal rule estimate of an integral

        parameter ( kases = 8 )

c       Shared variables .. available to all processors
        shared real     /reals/ sum, h, a, b
        shared integer  /ints/ id(32), Nfunc, Nprocs, N

        external adder
        integer procid(32)
        integer map(32)

        integer Npts( kases )
        integer clock(32,kases)
        integer maxPEs

c       Dialog with user : Set up experiments by getting a set of problem
c       sizes and computer card numbers on which to carry out the experiments:

    1   write(6,*) 'Maximum number of processors ?  0 to stop.'
        read(5,*) maxPEs
        if( maxPEs .le. 0 ) stop

        do 10 i = 1,maxPEs
                write(6,7) i
    7           format( 'Computer #', i2, ' = ?' )
                read(5,*) map(i)
   10   continue

        write(6,*) 'a = ?'
        read(5,*)  a
        write(6,*) 'b = ?'
        read(5,*)  b

   20   write(6,*) 'Number of cases to try = ?'
        read(5,*) Nkases
                if( Nkases .gt. kases ) then
                        write(6,*) 'Too many. Try again.'
                        go to 20
                endif


        do 30 i = 1, Nkases
                write(6,27) i
   27           format('How many evaluation points for case',i3,' ?')
                read(5,*) Npts(i)
   30   continue

c       Choice of three integrands:
        write(6,*) 'Function number = ?'
        read(5,*) Nfunc
```

```
c       Main experiment loop:
        do 900 nn = 1, Nkases
                N = Npts(nn)
                h = ( b-a ) / float(N)
        do 800 Nprocs = 1, maxPEs

        call CFrtic(istart)
        sum = 0.0

                COBLOCK
                do 50 i = 1, Nprocs
                        id(i) = i
                        itemp = map(i)
                        process( procid(i), adder(id(i)), itemp )
 50             continue
                END COBLOCK

        sum = h * ( .5*f(a,Nfunc) + sum + .5*f(b,Nfunc) )
        call CFrtic(istop)
        clock(Nprocs,nn) = istop  - istart
        write(6,*) 'Sum = ',sum

 800    continue


c       ================================================================
c       Print table giving performance results so far:
        write(6,812) Nfunc, ( Npts(i), i = 1, nn )
 812    format( 'PROBLEM 1: TRAPEZOIDAL RULE ESTIMATE ',
     1          'OF INTEGRAL. Function number ', i3, '.',/,/,
     2  'Number of              -> -> Number of evaluation points ->->',
     3    /, 'processors ', 10i8 )
        write(6,813)
 813    format( '            +-------------------------------------------',
     1  '-------------------------------------------' )

        do 850 i = 1, maxPEs
        write(6,837) i, ( clock(i,j), j = 1,nn )
 837    format( '  I', i3, 4x,'I', 10i8 )
        write(6,847) (float(clock(1,j)) /float(i*clock(i,j)),j=1,nn)
 847    format( '  v         I', 10f8.5 )
        write(6,848)
 848    format('             I')
 850    continue

c       ================================================================


 900    continue
        go to 1
        end
```

```
c       One copy of this subroutine is mapped to each processor designated
c       in the experiment:

        subroutine adder(myid)

        shared real     /reals/ sum, h, a, b
        shared integer  /ints/ id(32), Nfunc, Nprocs, N

c       Local variable:
        real temp, x

        temp = 0.0
        do 80 i = myid, N-1, Nprocs
                x = a + i * h
                temp = temp + f(x, Nfunc)
   80   continue

c       Sum is accumulated locally, so only one common memory access needed.
c       Note the need to insure exclusive access to global sum.

        call CFlock(ICFret,1,'sum')
        sum = sum + temp
        call CFulck(ICFret,1,'sum')

        return
        end



        real function f(x,n)
        If( n .le. 0 ) then
                f = x**2
        else
                If ( n .eq. 1 )  then
                        f = sin(x)
                else
                        f = exp(x)
                endIf
        endIf
        return
        end
```

```
c       Problems 2 and 3. Generalized E-STAR .. the sum of the products of
c       the elements in each row of a matrix.  Strategy: divide the rows up
c       as evenly as possible among the processors, each processor computing
c       the product for an entire row of the matrix.

        parameter ( kases = 8 )

        shared real     /reals/ sum
        shared integer  /ints/ id(32), Nfunc, Nprocs, N,M
        external worker

c       A process id and computer assignment for each spawned process.
        integer procid(32)
        integer map(32)

        integer Nsize( kases )
        integer Msize( kases )
        integer clock(32,kases)
        integer maxPEs


c       ================================================================
c       User input portion of program .. Specify the experiments to run.

    1   write(6,*) 'Maximum number of processors ?  0 to stop.'
        read(5,*) maxPEs
        If( maxPEs .le. 0 ) stop

        do 10 i = 1,maxPEs
                write(6,7) i
    7           format( 'Computer #', i2, ' = ?' )
                read(5,*) map(i)
   10   continue


   20   write(6,*) 'Number of cases to try = ?'
        read(5,*) Nkases
                If( Nkases .gt. kases ) then
                        write(6,*) 'Too many. Try again.'
                        go to 20
                endif


        do 30 i = 1, Nkases
                write(6,27) i
   27       format('Enter N and M for case',i3,' :')
                read(5,*) Nsize(i), Msize(i)
   30   continue


        write(6,*) 'Function number = ?'
        read(5,*) Nfunc

c       ================================================================
```

```
c       Loop over all experiments, collecting timing data:

        do 900 nn = 1, Nkases
                N = Nsize(nn)
                M = Msize(nn)

        do 800 Nprocs = 1, maxPEs

        call CFrtic(istart)
        sum = 0.0

                COBLOCK
                do 50 i = 1, Nprocs
                        id(i) = i
                        itemp = map(i)
                        process( procid(i), worker(id(i)), itemp )
  50            continue
                END COBLOCK

        call CFrtic(istop)
        clock(Nprocs,nn) = istop  - istart
        write(6,*) 'Sum = ' ,sum

  800   continue

c       ================================================================
c       Print table of performance results so far:

        write(6,812) Nfunc,(Nsize(i),i=1,nn)
  812   format( 'PROBLEMS 2 and 3 .. SUM OF PRODUCTS. '
     1  'Function number ', i3, '.',/,/,
     2  '       \   -> -> Size of problem ->->', /,
     3  '# of    \  N:', i6, 9i8  )
        write(6, 813 ) ( Msize(i), i = 1, nn )
  813   format( 'procs     \ M:', i6, 9i8 )
        write(6,814)
  814   format( '           +-------------------------------------------',
     1    '-------------------------------------------' )


        do 850 i = 1, maxPEs
        write(6,837) i, ( clock(i,j), j = 1,nn )
  837   format( '  |', i3, 4x,'|', 10i8 )
        write(6,847) (float(clock(1,j))/float(i*clock(i,j)),j=1,nn)
  847   format( '  v       |', 10f8.5 )
        write(6,848)
  848   format('          |')
  850   continue

c       ================================================================


  900   continue
        go to 1
        end
```

```
      subroutine worker(myid)

      shared real     /reals/ sum
      shared integer  /ints/ id(32), Nfunc, Nprocs, N,M

      real temp

      temp = 0.0
      do 80 i = myid, N, Nprocs
               prod = 1.0
               do 70 j = 1, M
                        prod = prod*f(i,j,Nfunc)
70             continue

               temp = temp + prod
80    continue

c      Accumulate sum locally, then do only one write to common memory.
c      WHEN construct insures exclusive access.

      when( sum .eq. sum ) then
      sum = sum + temp
      end when

      return
      end




      real function f(i,j,n)

      if( n .le. 1 ) then
               f = 1.0 + exp( -1.0 * float(abs(i-j)) )
      else
               f = 1.0001
      endif

      return
      end
```

```
c       Problem 4 .. Sum of inverses of elements in a vector.
c       Strategy: Divide the elements up among the processors ( as evenly
c       as posible). Each processor will invert and sum his elements,
c       and add this to the global sum.

        parameter ( kases = 8 )

        shared real    /reals/ sum, x(200000)
        shared integer /ints/ id(32), Nfunc, Nprocs, N
        external adder

        integer procid(32)
        integer map(32)

        integer Ndim( kases )
        integer clock(32,kases)
        integer maxPEs


c       ============================================================
c       User input section .. set up experiments on various vector
c       lengths and processor counts:

    1   write(6,*) 'Maximum number of processors ?  0 to stop.'
        read(5,*) maxPEs
        if( maxPEs .le. 0 ) stop

        do 10 i = 1,maxPEs
                write(6,7) i
    7           format( 'Computer #', i2, ' = ?' )
                read(5,*) map(i)
   10   continue


   20   write(6,*) 'Number of cases to try = ?'
        read(5,*) Nkases
                if( Nkases .gt. kases ) then
                        write(6,*) 'Too many. Try again.'
                        go to 20
                endif


        do 30 i = 1, Nkases
                write(6,27) i
   27           format('How many vector elements for case',i3,' ?')
                read(5,*) Ndim(i)
   30   continue

c       ============================================================
```

```
c       LOOP over experiments:

        do 900 nn = 1, Nkases
                N = Ndim(nn)

        do 800 Nprocs = 1, maxPEs

c       Reset vector:
        do 40 i = 1, N
                x(i) = 1.0/ (1.0 + float(i) )
  40    continue
                x(11) = 0.0


        call CFrtic(istart)
        sum = 0.0
                COBLOCK
                do 50 i = 1, Nprocs
                        id(i) = i
                        itemp = map(i)
                        process( procid(i), adder(id(i)), itemp )
  50            continue
                END COBLOCK
        call CFrtic(istop)
        clock(Nprocs,nn) = istop  - istart
        if(clock(Nprocs,nn) .le. 0 ) clock(Nprocs,nn) = 1

        write(6,*) 'Sum = ' ,sum  '
  800   continue



c       ================================================================
c       Print table of current performance results.

        write(6,812) ( Ndim(i), i = 1, nn )
  812   format( 'PROBLEM 4: MASKED SUMMATION.', /,/,
     1  '# of    \ ',20x,' -> -> Number of vector elements ->->', /,
     2  'procs    \', 10i8 )
        write(6,813)
  813   format( '          +-------------------------------------------',
     1  '-------------------------------------------' )


        do 850 i = 1, maxPEs
        write(6,837) i, ( clock(i,j), j = 1,nn )
  837   format( '  |', i3, 4x,'|', 10i8 )
        write(6,847) (float(clock(1,j))/float(i*clock(i,j)),j=1,nn)
  847   format( '  v        |', 10f8.5 )
        write(6,848)
  848   format('           |')
  850   continue
c       ================================================================


  900   continue
        go to 1
        end
```

```
      subroutine adder(myid)


      shared real     /reals/ sum, x(100000)
    shared integer  /ints/ id(32), Nfunc, Nprocs, N

      real temp

      temp = 0.0
      do 80 i = myid, N, Nprocs
            If( x(i) .ne. 0.0 ) temp = temp + 1.0/x(i)
80    continue

      call CFlock(ICFret,1,'sum')
      sum = sum + temp
      call CFulck(ICFret,1,'sum')

      return
      end




      real function f(x,n)


      If( n .le. 0 ) then
            f = x**2
      else
            If ( n .eq. 1 )  then
                  f = sin(x)
            else
                  f = exp(x)
            endIf
       endif


      return
      end
```

```
c        Problem 5 .. Compute statistics on a table of grades

         parameter(isizes=8)
         shared real /r/ score(300,300), avg(300),top(300),lowabv(300)
         shared integer /i1/ id(32), Nprocs, nabove(300)
         shared integer /i2/ Ntests, Nstuds
         shared logical /logvar/ genius
         external avcalc, stats


         real pi
         integer p(32),map(32),clock(32,isizes)
         integer itests(isizes),istuds(isizes)
         integer t1, t2

         data pi / 3.141592654 /


c        ================================================================
c        Problem definition phase: Consult with user to set up experiments.

   10    write(6,*) 'How many cases ?  0 to stop'
         read(5,*) kases
         if ( kases .gt. isizes ) then
                  write(6,*) 'TOO MANY. Must be <=', isizes
                go to 12
         endif

         if( kases .le. 0 ) stop


         do 15 i = 1, kases
   13         write(6,14) i
   14         format( '  Enter # tests and # students for case',i3,
    1                 '. Must be <= 300.' )
                  read(5,*)  itests(i), istuds(i)
              if ( (istuds(i) .gt. 300) .or. (itests(i) .gt. 300) ) then
              write(6,*) '              TOO LARGE. Try again.'
              go to 13
              endif
   15    continue


         write(6,*) 'Maximum number of processors for this experiment'
         read(5,*) nowPEs

         do 20 i = 1, nowPEs
                  write(6,17) i
   17        format('  Enter computer # for process',i3)
                  read(5,*) map(i)
   20    continue
c        ================================================================
```

```
c       Loop over experiments:

        do 1000 nn = 1, kases
                Ntests = itests(nn)
               Nstuds = istuds(nn)

        do 800 Nprocs = 1, nowPEs


c       INITIALIZE SCORE TABLE:
        do 40 i = 1, Ntests
        do 30 j = 1, Nstuds

                score(i,j) = 100.0 * sin( float(i+ j) )

   30   continue
   40   continue


c       First parallel phase:  compute average of each row:

                call CFrtic(t1)
                COBLOCK
                do 50 i = 1, Nprocs
                        id(i) = i
                        j = map(i)
                        process( p(i), avcalc(id(i)), j)
   50           continue
                END COBLOCK



c       Second parallel phase:
c       Distribute computations for each student among processors:

        genius = .false.
                COBLOCK
                do 100 i = 1, Nprocs
                        id(i) = i
                        j = map(i)
                        process( p(i), stats(id(i)), j )
  100           continue
                END COBLOCK

        call CFrtic(t2)
        clock(Nprocs,nn) = t2 - t1
        If( clock(Nprocs,nn) .le. 0 ) clock(Nprocs,nn) = 1

  800   continue


c       ==========================================================
c       Print table of results:

c                       — Code deleted for brevity —

c       ==========================================================

 1000   continue
        go to 10
        end
```

```
      subroutine avcalc( istart )

      shared real /r/ score(300,300), avg(300),top(300),lowabv(300)
      shared integer /i1/ id(32), Nprocs, nabove(300)
      shared integer /i2/ Ntests, Nstuds
      shared logical /logvar/  genius

      real temp
      do 250 i = istart,Ntests, Nprocs
            temp = 0.0

            do 200 j = 1, Nstuds
            temp = temp + score(i,j)
200         continue

            temp = temp  / float(Nstuds)
            avg(i) = temp
250   continue
      return
      end



      subroutine stats(jstart)

      shared real /r/ score(300,300), avg(300),top(300),lowabv(300)
      shared integer /i1/ id(32), Nprocs, nabove(300)
      shared integer /i2/ Ntests, Nstuds
      shared logical /logvar/  genius

      real ttemp, ltemp, stemp
      integer ntemp
      logical anygen,  allabv

      anygen = .false.
      do 350 j = jstart,Nstuds,Nprocs
            ntemp = 0
            ttemp = 0.0
            ltemp = 50000.0
            allabv = .true.

      do 300 i = 1, Ntests
            stemp = score(i,j)
            if( stemp .gt. ttemp ) ttemp = stemp
            if( stemp .gt. avg(i) ) then
              if( stemp .lt. ltemp ) ltemp = stemp
                  ntemp = ntemp + 1
                  score(i,j) = 1.1*stemp
            else
                  allabv = .false.
            endif
300   continue
            anygen = anygen .or. allabv
            top(j) = ttemp
            lowabv(j) = ltemp
            nabove(j) = ntemp
350   continue
      genius = genius .or. anygen

      return
      end
```

```
c       Problem 6
c       Tridiagonal solver using iterative method of Jordan.

        parameter ( kases = 8 )
        shared real /reals1/ u(10000,2),l(10000,2), limit
        shared real /reals2/ d(10000,2),x(10000),y(10000,2)
        shared integer /ints/ id(32),Nprocs,N,isynch
        external worker

        integer procid(32)
        integer map(32)

        integer Nsize( kases )
        integer clock(32,kases)
        integer maxPEs

c       ================================================
c       User dialog phase. Get computer numbers and
c       problem sizes for the various cases.
c             --- Code deleted for brevity ---
c       ================================================

c       LOOP OVER EXPERIMENTS:
        do 900 nn = 1, Nkases
                N = Nsize(nn)
                limit = 1.44269504 * alog( float( N ) ) + .1
        do 800 Nprocs = 1, maxPEs

c       Initialize vectors:
        do 40 i = 1,N
                l(i,1) = 1.0
                d(i,1) = 1.0
                u(i,1) = 1.0
                y(i,1) = 15.0
   40   continue
        y(1,1) = 10.0
        y(N,1) = 10.0

        call CFrtic(istart)
        isynch = 0
                COBLOCK
                do 50 i = 1, Nprocs
                        id(i) = i
                        itemp = map(i)
                        process( procid(i), worker(id(i)), itemp )
   50           continue
                END COBLOCK
        call CFrtic(istop)
        clock(Nprocs,nn) = istop  - istart

        if( (Nprocs .eq. 1) .and. ( N .le. 50 ) ) then
                do 80 i = 1, N
                write(6,*) 'x(',i,') = ', x(i), y(i,1),d(i,1)
   80           continue
        endif

  800   continue

c       ================================================
c       Print table of performance results.  Code omitted.
c       ================================================

  900   continue
        go to 1
        end
```

```
c       Each computer executes a copy of this routine. Note that since
c       this is an iterative method, synchronization is necessary to
c       insure that all processors are on the same iteration. Global
c       integer values serve as synchronization semaphores.

        subroutine worker(myid)

        shared real /reals1/ u(10000,2),l(10000,2), limit
        shared real /reals2/ d(10000,2),x(10000),y(10000,2)
        shared integer /ints/ id(32),Nprocs,N,isynch

c       TOP OF ITERATION LOOP:
   10   continue

c       Usual update is performed on a subset of vector elements:
        do 50 i = myid,N,Nprocs
                if( d(i,1) .ne. 0.0 ) then
                        l(i,2) = l(i,1)/d(i,1)
                        u(i,2) = u(i,1)/d(i,1)
                        y(i,2) = y(i,1)/d(i,1)
                endif
   50   continue

c       ------------------------------------------------------------
c       SYNCHRONIZATION: Wait here for fellow processors.
        call CFlock(ICFret,1,'isynch')
                isynch = isynch + 1
        call CFulck(ICFret,1,'isynch')
        when(isynch .ge. (2*mycntr-1)*Nprocs ) continue
c       ------------------------------------------------------------

        do 100 i = myid,N,Nprocs
        d(i,1) = 1. - l(i,2)*u(i-k,2) - u(i,2)*l(i+k,2)
        y(i,1) = y(i,2) - l(i,2)*y(i-k,2) - u(i,2)*l(i+k,2)
        l(i,1) = -l(i,2)*l(i-k,2)
        u(i,1) = u(i,2)*u(i+k,2)
  100   continue

c       ------------------------------------------------------------
c       SYNCHRONIZATION: Again wait for fellows.
        call CFlock(ICFret,1,'isynch')
                isynch = isynch + 1
        call CFulck(ICFret,1,'isynch')
        when( isynch .ge. 2*Nprocs*mycntr ) continue
c       ------------------------------------------------------------

        mycntr = mycntr + 1
        k = 2*k
        if( mycntr .gt. limit ) then
                do 150 i = myid,N,Nprocs
                x(i) = y(i,1)/d(i,1)
  150           continue
                return
        endif

        go to 10
        end
```

```
c       Problem 7.  Compute polynomial interpolant values at 5 points
c       using the Lagrange interpolation formulas.

        parameter ( kases = 8 )

        shared real     /reals/ sum(5), x(5), node(10000)
        shared integer  /ints/ id(32), Nprocs, N
        external adder

*       integer procid(32)
        integer map(32)

        integer Ndim( kases )
        integer clock(32,kases)
        integer maxPEs

c       ================================================================
c       Query user to establish experimental parameters. Set up x(1 .. 5). Code omitted.
c       ================================================================

c       LOOP OVER EXPERIMENTS:
        do 900 nn = 1, Nkases
                N = Ndim(nn)

        do 40 i = 1, N
                node(i) = .08 * i
 40     continue

        do 800 Nprocs = 1, maxPEs

        call CFrtic(istart)

        do 45 i = 1,5
        sum(i) = 0.0
 45     continue

                COBLOCK
                do 50 i = 1, Nprocs
                        id(i) = i
                        itemp = map(i)
                        process( procid(i), adder(id(i)), itemp )
 50             continue
                END COBLOCK

        call CFrtic(istop)
        clock(Nprocs,nn) = istop  - istart
        if(clock(Nprocs,nn) .le. 0 ) clock(Nprocs,nn) = 1

800     continue

        write(6,55) maxPEs, Ndim(nn)
 55     format(/,'---------- Answer computed by ', i3, ' procs for ', i5, ' nodes.' )
        do 60 i = 1, 5
        write(6,58) i,sum(i), f(x(i)), abs( (f(x(i))-sum(i))/f(x(i)) )
 58     format( 'sum(',i2,') = ',e12.5,' actual = ',e12.5,
     1          ' rel err =',e12.5)
 60     continue

c       ================================================================
c       Print out performance results for the experiments done so far. Code deleted for brevity
c       ================================================================

900     continue
        go to 1
        end
```

```
c       Each computer executes a copy of this. The job is to compute an equal
c       number of the terms in the Lagrange sum, and add these into a global variable.

        subroutine adder(myid)

        shared real     /reals/ sum(5), x(5), node(10000)
        shared integer  /ints/ id(32), Nprocs, N

c       LOOP over the 5 summations.
        do 300 k = 1,5
           temp = 0.0

           do 200 i = myid,N,Nprocs
                  prod = 1.0
                  do 100 j = 1,N
                  if (j .ne. i) prod = prod*(x(k)-node(j))/(node(i)-node(j))
 100              continue

                  temp = temp + f( node(i) ) * prod

 200       continue

c       ------------------------------------------------------------
c       Write result into shared memory, using WHEN for exclusive access
        call CFlock(ICFret,1,'sum')
        sum(k) = sum(k) + temp
        call CFulck(ICFret,1,'sum')
c       ------------------------------------------------------------

 300    continue
        return
        end



        real function f(x)
        f = exp(x)
        return
        end
```

```
c          Problem 8.  Construct divided difference table.

           parameter ( kases = 8 )

           shared real     /reals/ x(300),d(300,300)
           shared integer  /ints/ id(32), Nprocs, N, mypos(32)
           external worker

           integer procid(32)
           integer map(32)

           integer Ndim( kases )
           integer clock(32,kases)
           integer maxPEs



c          ========================================
c          Query users for parameters of experiments. Code omitted.
c          ========================================

c          LOOP OVER EXPERIMENTS:
           do 900 nn = 1, Nkases
                     N = Ndim(nn)

           do 40 i = 1,N
                     x(i) = .2 * i + .01 * cos(float(i))
                     d(i,1) = f(x(i))
   40      continue

           do 800 Nprocs = 1, maxPEs

           do 45 i = 1,Nprocs
                     mypos(i) = 0
   45      continue
           mypos(Nprocs) = N

           call CFrtic(istart)
                     COBLOCK
                     do 50 i = 1, Nprocs
                               id(i) = i
                               itemp = map(i)
                               process( procid(i), worker(id(i)), itemp )
   50                continue
                     END COBLOCK
           call CFrtic(istop)
           clock(Nprocs,nn) = istop  - istart
           if(clock(Nprocs,nn) .le. 0 ) clock(Nprocs,nn) = 1

  800      continue

           if( N .le. 10) then
                     do 80 i = 1,N
                     write(6,77) x(i),(d(i,k),k=1,N-k+1)
   77                format(11f7.3)
   80      continue
           endif

c          ================================================================
c          Display performance results which have been collected so far. Code omitted.
c          ================================================================
  900      continue
           go to 1
           end
```

```
c       Columns are divided among processors, each processor constructing the whole
c       column. Each processor is synchronized with the processor working to his left,
c       working one row higher (because of the data dependencies). Synchronization
c       overhead is extremely high. Each entry in the table is associated with a
c       number, specifically, (j-1)*N + i for the (i,j) entry. Each pocessor
c       records the number of the entry he most recently computed in an array in shared
c       memory called "mypos", so that processors to his right may synchronize on him.

        subroutine worker(myid)

        shared real     /reals/ x(300),d(300,300)
        shared integer  /ints/ id(32), Nprocs, N, mypos(32)

c       This is the proc. id. of the processor operating to my immediate left.
        integer master
        master = Nprocs - mod(Nprocs-myid+1, Nprocs)

        do 200 k = myid+1,N,Nprocs

                when(mypos(master) .ge. (k-2)*N + 2) continue
                d(1,k) = ( d(2,k-1)-d(1,k-1) )/ (x(1+k)-x(1))
                mypos(myid) = N*(k-1) + 1

                do 100 i = 2,N-k+1
                when(mypos(master) .ge. mypos(myid)-N+1) continue
                d(i,k) = ( d(i+1,k-1)-d(i,k-1) )/ (x(i+k)-x(i))
                mypos(myid) = mypos(myid) + 1
100             continue

200     continue

        return
        end




        real function f(x)
        f = sin(x)
        return
        end
```

```
c       Problem 9
c       Image smoothing on an N by M grid. Mild synchronization.

        parameter ( kases = 8 )

        shared real     /reals/ a(300,300,2)
        shared integer  /ints/ id(32), Nprocs, N, isynch, K
        external worker

        integer procid(32)
        integer map(32)

        integer Ndim( kases )
        integer reps( kases )
        integer clock(32,kases)
        integer maxPEs

c       ------------------------------------------------------
c       User dialog phase.  Read in experimental parameters.  Code omitted
c       ------------------------------------------------------


        LOOP OVER EXPERIMENTS:
        do 900 nn = 1, Nkases
                N = Ndim(nn)
                K = reps(nn)

        do 800 Nprocs = 1, maxPEs

        do 45 i = 1,N
        do 40 j = 1,N
                a(i,j,1) = 125.0 * sin(float(i+j))
 40     continue
 45     continue

        isynch = 0
        call CFrtic(istart)
                COBLOCK
                do 50 i = 1, Nprocs
                        id(i) = i
                        itemp = map(i)
                        process( procid(i), worker(id(i)), itemp )
 50             continue
                END COBLOCK
        call CFrtic(istop)
        clock(Nprocs,nn) = istop  - istart
        if(clock(Nprocs,nn) .le. 0 ) clock(Nprocs,nn) = 1

 800    continue

        if( N .le. 10) then
                do 80 i = 1,N
                itog = 1 + mod(K,2)
                write(6,77) (a(i,j,itog),j=1,N)
 77             format(10f8.3)
 80             continue
        endif

c       ------------------------------------------------
c       Print performance results so far  Code omitted.
c       ------------------------------------------------

 900    continue
        go to 1
        end
```

```
c       Rows are divided among the processors. Each processor handles an entire row.
c       Synchronization is necessary at the end of each pass over the grid to insure
c       all processors are working on the same pass.

        subroutine worker(myid)

        shared real     /reals/ a(300,300,2)
        shared integer  /ints/ id(32), Nprocs, N, isynch, K

        itog = 1
        do 500 mycntr = 1,K

                iold = itog
                itog = 1 + mod(itog,2)

c               ---------------------------------------------------------
c               Wait here on fellows:
                when( isynch .ge. Nprocs*(mycntr-1)) continue
c               ---------------------------------------------------------

                do 300 i = myid+1,N-1,Nprocs
                do 200 j = 2,N-1

                a(i,j,itog) = ( a(i-1,j,iold) + a(i+1,j,iold) +
     1                                  a(i,j-1,iold) + a(i,j+1,iold) )/4.0

  200           continue
  300           continue

c               ---------------------------------------------------------
c               Notify semaphore that I'm done.
                call CFlock(ICFret,1,'isynch')
                isynch = isynch + 1
                call CFulck(ICFret,1,'isynch')
c               ---------------------------------------------------------

  500   continue


        return
        end
```

```
c          Problem 10. LU factorization by Gaussian elimination with partial pivoting.

           parameter ( kases = 8 )

           shared real     /reals/ a(300,300), pivot
         shared integer  /ints/ id(32), Nprocs, N, isynch,ipvt,row(300)
           external worker

           integer procid(32)
           integer map(32)
           integer Ndim( kases )
           integer clock(32,kases)
           integer maxPEs

c          ================================================
c          Query user for experiment sizes, computer numbers, etc. Code omitted.
c          ================================================

c          LOOP OVER EXPERIMENTS:
           do 900 nn = 1, Nkases
                    N = Ndim(nn)
           do 800 Nprocs = 1, maxPEs

           do 45 i = 1,N
                    row(i) = i
           do 40 j = 1,N
                    a(i,j) = 125.0 * sin(float(i+j))
   40      continue
   45      continue

c          Go ahead and (sequentially) find the first pivot.
           call CFrtic(istart)
           ipvt = 1
           pivot = 0.0
           do 47 i = 1,N
                    if( abs(a(i,1)) .gt. pivot) then
                    pivot = abs(a(i,1))
                    ipvt = i
                    endif
   47      continue

                    isynch = Nprocs -1
                    COBLOCK
                    do 50 i = 1, Nprocs
                             id(i) = i
                             itemp = map(i)
                             process( procid(i), worker(id(i)), itemp )
   50               continue
                    END COBLOCK

           call CFrtic(istop)
           clock(Nprocs,nn) = istop  - istart
           if(clock(Nprocs,nn) .le. 0 ) clock(Nprocs,nn) = 1

   800     continue
c          ================================================================
c          Display current performance results, and factored matrix if small. Code omitted.
c          ================================================================
   900     continue
           go to 1
           end
```

```
      subroutine worker(myid)

      shared real    /reals/ a(300,300), pivot
      shared integer /ints/ id(32), Nprocs, N, isynch,ipvt,row(300)

c     ================================================================
c     LOOP OVER SUBMATRIX STEPS:  At the beginning of each such step, it is assumed
c     that the pivot element was determined on the previous step, as that submatrix
c     was computed. (This explains the initialization in the main program).
c     ================================================================

      do 500 k = 1,N

c                 Process 1 actually makes the swap.
                  if( myid .eq. 1 ) then
c                 ------------------------------------------------
c                   Wait until all procs are done with last submatrix step.
                    when( isynch .ge. k*Nprocs - 1 ) continue
c                 ------------------------------------------------
                    itemp = row(k)
                    row(k) = row(ipvt)
                    row(ipvt) = itemp

                    pivot = 0.0
                    ipvt = k+1
                  endif

c                 ------------------------------------------------
c                 Signal semaphore and wait here for fellow processors.
                  call CFlock(ICFret,1,'isynch')
                  isynch = isynch + 1
                  call CFulck(ICFret,1,'isynch')
                  when( isynch .ge. k*Nprocs ) continue

c                 ------------------------------------------------
c                 Rows of submatrix are distributed among processors. Compute next pivot
c                 by looking at the entries in the (k+1)st column as they are computed.
                  do 400 i = k+myid, N, Nprocs

                  a(row(i),k) = a(row(i),k) /a(row(k),k)
                  do 300 j = k+1,N
                  a(row(i),j) = a(row(i),j) - a(row(i),k)*a(row(k),k)
  300             continue

c                   ------------------------------------------------
                    call CFlock(ICFret,1,'pivot')
                    if( abs(a(row(i),k+1)) .gt. pivot ) then
                    pivot = abs(a(row(i),k+1))
                    ipvt = i
                    endif
                    call CFulck(ICFret,1,'pivot')
c                   ------------------------------------------------

  400             continue

  500             continue


      return
      end
```

```
c       Problem II. Filter data in a vector and write to a database.

        parameter ( kases = 8 )

        shared real /reals/ data(100000), sum(4)
        shared integer /ints/ N, id(32), Nprocs
        external worker

        integer procid(32)
        integer map(32)

        integer Ndim( kases )
        integer clock(32,kases)
        integer maxPEs

c       ============================================================
c       Query user for experiment sizes, computer numbers, etc. Code omitted.
c       ============================================================

        do 900 nn = 1, Nkases
                N = Ndim(nn)
        do 800 Nprocs = 1, maxPEs

        do 40 i = 1, N
                data(i) =  -float(i)/.1 + 1080.0 * sin(float(i+10))
  40    continue

        call CFrtic(istart)
        do 45 i = 1,4
        sum(i) = 0.0
  45    continue

                COBLOCK
                do 50 i = 1, Nprocs
                        id(i) = i
                        itemp = map(i)
                        process( procid(i), worker(id(i)), itemp )
  50            continue
                END COBLOCK

c       Save data in database within timer scope:
        do 60 i = 1,4
                write(6,*) sum(i)
  60    continue

        call CFrtic(istop)
        clock(Nprocs,nn) = istop  - istart
        if(clock(Nprocs,nn) .le. 0 ) clock(Nprocs,nn) = 1

 800    continue
c       ============================================
c       Here, print table of performance results
c       ============================================
 900    continue
        go to 1
        end
```

```
c       Perform filter on select vector elements, acummulating sums, and then
c       add this to the shared sums.

        subroutine worker(ident)
        shared real /reals/ data(100000), sum(4)
        shared integer /ints/ N, id(32), Nprocs

        real locsum(4), pi, temp
        data pi / 3.141592654 /

        do 50 i = 1, 4
                locsum(i) = 0.0
50      continue

        do 100 i = ident, N, Nprocs

                temp = amax1( 0.0, amin1( 1000.0, data(i) ) )
                data(i) = temp

                locsum(1) = locsum(1) + temp

                do 75 k = 2, 4
                locsum(k) = locsum(k) + cos( float(pi*k*i) /float(N+1))*temp
75              continue

100     continue



c       Note that in this block, access to SUM entries will
c       be exclusive.

        call CFlock(ICFret,1,'sum')
                sum(1) = sum(1) + locsum(1)/N
                sum(2) = sum(2) + locsum(2)
                sum(3) = sum(3) + locsum(3)
                sum(4) = sum(4) + locsum(4)
        call CFulck(ICFret,1,'sum')

        return
        end
```

```
c       Problem 12. Simple data movement. Move data from smaller array to form
c       a composite larger one.

        parameter ( kases = 8 )

        shared real    /r1 / a(300,300), c(300), r(300)
        shared real    /r2 / acom, abig(300,300)
        shared integer /ints/ id(32), Nprocs, N,M
        external worker

        integer procid(32)
        integer map(32)

        integer Nsize( kases )
        integer Msize( kases )
        integer clock(32,kases)
        integer maxPEs

c       ================================================
c       Query user for experiment sizes, computer numbers, etc. Code omitted.
c       ================================================

        do 900 nn = 1, Nkases
                N = Nsize(nn)
                M = Msize(nn)

                do 40 i = 1,N
                        c(i) = float(1+i)
                do 35 j = 1,M
                        a(i,j) = float(i+j)
35              continue
40              continue
                do 45 j = 1,M
                        r(j) = float(1-j)
45              continue
                acom = .5

        do 800 Nprocs = 1, maxPEs

        call CFrtic(istart)
        COBLOCK
                do 50 i = 1, Nprocs
                        id(i) = i
                        itemp = map(i)
                        process( procid(i), worker(id(i)), itemp )
50              continue
        END COBLOCK

        abig(N+1,M+1) = acom
        call CFrtic(istop)
        clock(Nprocs,nn) = istop  - istart
800     continue

c       ========================
c       Print table of performance results
c       ========================

900     continue
        go to 1
        end
```

```
c          Move entries into select rows of the big matrix.

           subroutine worker(myid)

           shared real      /r1 / a(300,300), c(300), r(300)
           shared real      /r2 / acom, abig(300,300)
         shared integer  /ints / id(32), Nprocs, N,M


           do 100 i = myid,N,Nprocs
                      abig(i,M+1) = c(i)
           do  50 j = 1,M
                      abig(i,j) = a(i,j)
  50       continue
 100       continue

           do 200 j = myid,M,Nprocs
                      abig(N+1,j) = r(j)
 200       continue


           return
           end
```

```
      parameter ( kases = 8 )

      shared real /reals/ sum, a(50000),b(50000),c(50000),d(50000)
      shared integer /ints/ id(32), Nprocs, N
      external adder

      integer procid(32)
      integer map(32)

      integer Ndim( kases )
      integer clock(32,kases)
      integer maxPEs


c     Here, read in experiment sizes, computer numbers, etc. from user. Code omitted.

      do 900 nn = 1, Nkases
            N = Ndim(nn)

      do 800 Nprocs = 1, maxPEs

      do 40 i = 1, N
            a(i) = i/float(10) + 1.0/float(i)
            b(i) = alog(a(i)) + .02
            c(i) = (a(i) + b(i)) * sin(a(i))
            d(i) = a(i) + b(i) - 2*c(i)
 40   continue

      call CFrtic(istart)
      sum = 0.0
      COBLOCK
            do 50 i = 1, Nprocs
                  id(i) = i
                  itemp = map(i)
                  process( procid(i), adder(id(i)), itemp )
 50         continue
      END COBLOCK
      call CFrtic(istop)
      clock(Nprocs,nn) = istop  - istart
      if(clock(Nprocs,nn) .le. 0 ) clock(Nprocs,nn) = 1

      write(6,*) 'Sum = ' ,sum
 800  continue

c     Here, write out intermediate performance results.

 900  continue
      go to 1
      end
```

```
      subroutine adder(myid)

      shared real /reals/ sum, a(50000),b(50000),c(50000),d(50000)
     shared integer /ints/ id(32), Nprocs, N

      real temp

      temp = 0.0
      do 80 i = myid, N, Nprocs

            a(i) = a(i)**sin(b(i))
            if( sin(a(i)) .lt. cos(c(i)) ) then
                  a(i) = a(i) + c(i)
            else
                  a(i) = a(i) - d(i)
            endif

            temp = temp + a(i)**2

80    continue

c     ---------------------------------------------------
c      CRITICAL REGION: Exclusively read and update SUM
      call CFlock(ICFret,1,'sum')
      sum = sum + temp
      call CFulck(ICFret,1,'sum')
c     ---------------------------------------------------

      return
      end




      real function f(x,n)


      if( n .le. 0 ) then
            f = x**2
      else
            if ( n .eq. 1 )  then
                  f = sin(x)
            else
                  f = exp(x)
            endif
       endif


      return
      end
```

```
c       Problem 14. Integration tests.

        parameter ( kases = 8 )

        shared real     /reals1/ sum(3,3), a, b
        shared real     /reals2/ h,hgauss,offset
        shared integer  /ints/ id(32), Nfunc, Nprocs, N, Ngauss

        external adder
        integer procid(32)
        integer map(32)
        real exact(3)
        integer Ndim( kases )
        integer clock(32,kases)
        integer maxPEs

c       ================================================
c       Here, read in parameters for experiments from user. This includes the
c       accuracies (#of evaluations), exact solution values with which a
c       relative error may be computed, the limits of integration a and b,
c       and the computer numbers.   CODE OMITTED.
c       ================================================

c       LOOP OVER EXPERIMENTS:
        do 900 nn = 1, Nkases
                N = Ndim(nn)
                h = (b-a)/float(N)
                Ngauss =  N/3
                hgauss =  (b-a)/float(Ngauss)
                offset =  .774596669241 * hgauss  / 2.0

        do 800 Nprocs = 1, maxPEs

        call CFrtic(istart)
        do 45 i = 1,3
        do 40 j = 1,3
                sum(i,j) = 0.0
   40   continue
   45   continue
                COBLOCK
                do 50 i = 1, Nprocs
                        id(i) = i
                        itemp = map(i)
                        process( procid(i), adder(id(i)), itemp )
   50           continue
                END COBLOCK


c       ADJUST SUMS:
        do 70 j = 1,3
        sum(1,j) = h*(sum(1,j) + f(a,j)/2.0 + f(b,j)/2.0 )
        sum(2,j) = h*(sum(3,j) + f(a,j) + f(b,j) )/3.0
        sum(3,j) = hgauss * sum(3,j)/ 18.0
   70   continue

        call CFrtic(istop)
        clock(Nprocs,nn) = istop  - istart
        if(clock(Nprocs,nn) .le. 0 ) clock(Nprocs,nn) = 1

  800   continue
c       ================================================
c       Print table of performance results.  Code omitted.
c       ================================================
  900   continue
        go to 1
        end
```

```
      subroutine adder(myid)

      shared real    /reals1/ sum(3,3), a, b
      shared real    /reals2/ h,hgauss,offset
      shared integer /ints/ id(32), Nfunc, Nprocs, N, Ngauss

      do 1000 Nfunc = 1,3

c     ------------------------------------------------------------
c     Compute temporary sum for trapezoidal rule.
      temp1 = 0.0
      do 100 i = myid,N-1,Nprocs
            x = i*h
            temp1 = temp1 + f(x,Nfunc)
  100 continue
c     ------------------------------------------------------------
c     Compute sum for Simpson's Rule.
      temp2 = 0.0
      do 200 i = myid,N-1,Nprocs
            x = i*h
            if( mod(i,2) .ge. 1 ) then
                  temp2 = temp2 + 4.0*f(x,Nfunc)
            else
                  temp2 = temp2 + 2.0*f(x,Nfunc)
            endif
  200 continue
c     ------------------------------------------------------------
c     Gaussian quadrature.
      temp3 = 0.0
      do 300 i = myid,Ngauss,Nprocs
            x = ( float(i) - .5 ) * hgauss
            temp3 = temp3 + 5.0*f(x-offset,Nfunc)
     1            + 8.0 * f(x,Nfunc) + 5.0 * f(x+offset,Nfunc)
  300 continue
c     ------------------------------------------------------------
c     Access shared memory sums, using system locks.
      call CFlock(ICFret,1,'sum')
            sum(1,Nfunc) = sum(1,Nfunc) + temp1
            sum(2,Nfunc) = sum(2,Nfunc) + temp2
            sum(3,Nfunc) = sum(3,Nfunc) + temp3
      call CFulck(ICFret,1,'sum')
c     ------------------------------------------------------------

 1000 continue
      return
      end


      real function f(x,n)
      if( n .le. 1 ) then
            f = exp(x)
      else
            if ( n .eq. 2 ) then
                  f = sqrt( abs(x-.2345) )
            else
                  f = 1.0 + x**2 + 1.0/( 1.0 + 100*x**2)
            endif
      endif
      return
      end
```

```
      parameter ( kases = 8 )
      shared real /reals/ a,b,sum,x(0:50001,2),t(10),approx(10,2)
      shared integer /ints/ id(32), Nprocs, N, K, isynch
      external adder

      integer procid(32)
      integer map(32)
      integer Ndim( kases )
      integer clock(32,kases)
      integer maxPEs
c     ================================================================
c     Query user for usual problem parameters, plus a vector t(1,..,K) of test points. Code omitted.
c     ================================================================

c     LOOP OVER EXPERIMENTS:
      do 900 nn = 1, Nkases
            N = Ndim(nn)
      do 800 Nprocs = 1, maxPEs

      call CFrtic(istart)
      do 40 j = 1,2
            x(0,j) = a - .1
            x(N+1,j) = b + .1
      do 38 i = 1,K
            approx(i,j) = 0.0
 38   continue
 40   continue

      isynch = 0
      COBLOCK
            do 50 i = 1, Nprocs
                  id(i) = i
                  itemp = map(i)
                  process( procid(i), adder(id(i)), itemp )
 50         continue
      END COBLOCK

      do 60 i = 1,K
      do 58 j = 1,2
            approx(i,j) = abs( (f(t(i)) - approx(i,j))/f(t(i)) )
 58   continue
 60   continue

      call CFrtic(istop)
      clock(Nprocs,nn) = istop - istart
      If(clock(Nprocs,nn) .le. 0 ) clock(Nprocs,nn) = 1

 800  continue
      write(6,805)
 805  format(//' t point  rel error  rel error',
     1                  '  uniform   chebyshev' )
      do 810 i = 1, K
            write(6,808) t(i),approx(i,1),approx(i,2)
 808        format(3e10.2)
 810  continue
c     ========================================
c     Here, print performance results. CODE OMITTED.
c     ========================================

 900  continue
      go to 1
      end
```

```
      subroutine adder(myid)

      shared real /reals/ a,b,sum,x(0:50001,2),t(10),approx(10,2)
      shared integer /ints/ id(32), Nprocs, N, K, isynch

      real h,pi
      data pi /3.141592654 /
      h = (b-a)/float(n-1)

      do 100 i = myid,N,Nprocs
               x(i,1) = a + (i-1)*h
               x(i,2) = a + (a-b)*cos(float( (2*i-1)*pi/(2*N) ) )
  100 continue

c     ---------------------------------------------
c      SYNCHRONIZATION...  Wait on fellows
      call CFlock(ICFret,1,'isynch')
      isynch = isynch + 1
      call CFulck(ICFret,1,'isynch')
      when( isynch .ge. Nprocs ) continue
c     ---------------------------------------------

      do 500 kk = 1,K
      temp1 = 0.0
      temp2 = 0.0

      do 400 i = myid,N,Nprocs
      temp1 = temp1 + f(x(i,1))       * hermc (t(kk),i,1) +
     1                fprime(x(i,1)) * hermc1(t(kk),i,1)
      temp2 = temp2 + f(x(i,2))       * hermc (t(kk),i,2) +
     1                fprime(x(i,2)) * hermc1(t(kk),i,2)
  400 continue

      call CFlock(ICFret,1,'approx')
               approx(kk,1) = approx(kk,1) + temp1
               approx(kk,2) = approx(kk,2) + temp2
      call CFulck(ICFret,1,'approx')

  500 continue
      return
      end
```

```
real function hermc(tpt,i,iflag)
shared real /reals/ a,b,sum,x(0:50001,2),t(10),approx(10,2)
shared Integer /ints/ id(32), Nprocs, N, K, isynch

If( (tpt .le. x(i-1,iflag)) .or. (tpt .ge. x(i+1,iflag) ) ) then
        hermc = 0.0
        return
endif

If( tpt .gt. x(i,iflag) ) then
        dt = x(i+1,iflag) - x(i,iflag)
        dx = x(i+1,iflag) - tpt
else
        dt = x(i,iflag) - x(i+1,iflag)
        dx = tpt - x(i,iflag)
endif

hermc = ( 3.0 - 2.0 *dx/dt ) * dx**2/dt**2
return
end
```

```
real function hermc1(tpt,i,iflag)
shared real /reals/ a,b,sum,x(0:50001,2),t(10),approx(10,2)
shared Integer /ints/ id(32), Nprocs, N, K, isynch

If( (tpt .le. x(i-1,iflag)) .or. (tpt .ge. x(i+1,iflag) ) ) then
        hermc1 = 0.0
        return
endif
dx = tpt - x(i,iflag)
If( tpt .gt. x(i,iflag) ) then
        dt2 = ( x(i,iflag) - x(i+1,iflag) )**2
        dx2 = ( tpt - x(i+1,iflag) )**2
else
        dt2 = ( x(i,iflag) - x(i-1,iflag) )**2
        dx2 = ( tpt - x(i-1,iflag) )**2
endif

hermc1 = dx2*dx/dt2
return
end
```

```
real function f(x)
f = x**2 + 25.0
return
end
```

```
real function fprime(x)
fprime = 2.0 * x
return
end
```

```
c       Problem 16. Factor and backsolve using a Hilbert matrix.

        parameter ( kases = 8 )

        shared real    /r1 / a(100,100),hilb(100,100),rnorm(4)
        shared real    /r2 / b(100,4), x(100,4), pivot, resid(100,4)
        shared integer /ints / id(32), Nprocs, N, isynch,ipvt,row(100)
        shared logical /log / done(100,2)
        external factor, solve

        integer procid(32)
        integer map(32)
        integer Ndim( kases )
        integer clock(32,kases)
        integer maxPEs

c       ================================================
c       Here, query user for problem sizes, computer numbers, etc.
c       ================================================

        do 35 i = 1,100
                b(i,1) = 0.0
                b(i,2) = 1.0
                b(i,3) = 1.0 + .01 * sin( float( 100*i) )
                b(i,4) = 0.0
        do 34 j = 1,100
                hilb(i,j) = 1.0/float(i+j-1)
                b(i,4) = b(i,4) + hilb(i,j)
   34   continue
   35   continue


c       LOOP OVER EXPERIMENTS:
        do 900 nn = 1, Nkases
                N = Ndim(nn)
        do 800 Nprocs = 1, maxPEs

        do 45 i = 1,N
                row(i) = i
                done(i,1) = .false.
                done(i,2) = .false.
        do 39 j = 1,4
                x(i,j) = b(i,j)
                resid(i,j) = b(i,j)
   39   continue
        do 40 j = 1,N
                a(i,j) = hilb(i,j)
   40   continue
   45   continue

        call CFrtic(istart)
        isynch = Nprocs -1
        ipvt = 1
        pivot = 0.0

        do 47 i = 1,N
                if( abs(a(i,1)) .gt. pivot) then
                pivot = abs(a(i,1))
                ipvt = i
                endif
   47   continue
```

```
c                   In this COBLOCK, the matrix "a" is factored in parallel.
                    COBLOCK
                    do 50 i = 1, Nprocs
                              id(i) = i
                              itemp = map(i)
                              process( procid(i), factor(id(i)), itemp )
   50               continue
                    END COBLOCK


c                   In this COBLOCK, backsolving and residual computation is done in parallel.
                    isynch = 0
                    COBLOCK
                    do 60 i = 1, Nprocs
                              id(i) = i
                              itemp = map(i)
                              process( procid(i),solve(id(i)), itemp )
   60               continue
                    END COBLOCK

              call CFrtic(istop)
              clock(Nprocs,nn) = istop  - istart
              If(clock(Nprocs,nn) .le. 0 ) clock(Nprocs,nn) = 1

   800        continue

c             ================================================================
c             Here, print table of performance results so far, and print solution vector.
c             ================================================================

   900        continue
              go to 1
              end



              subroutine factor(myid)
              shared real      /r1 / a(100,100),hilb(100,100),rnorm(4)
              shared real      /r2 / b(100,4), x(100,4), pivot, resid(100,4)
              shared integer   /ints / id(32), Nprocs, N, isynch,ipvt,row(100)
              shared logical   /log / done(100,2)


c                 - -   SEE CODE FOR ROUTINE "WORKER" OF PROBLEM 10   - -
```

```
      subroutine solve(myid)
      shared real     /r1/ a(100,100),hilb(100,100),rnorm(4)
      shared real     /r2/ b(100,4), x(100,4), pivot, resid(100,4)
      shared integer  /ints/ id(32), Nprocs, N, isynch,ipvt,row(100)
      shared logical  /log/ done(100,2)
      real temp(4)

c     F O R W A R D   S U B S T I T U T I O N
      do 500 i = myid,N,Nprocs
      do 400 j = 1,i-1
               when( done(j,1) ) continue
               do 40 k = 1,4
               x(row(i),k) = x(row(i),k) - a(row(i),j)*x(row(j),k)
 40            continue
400   continue
      done(i,1) = .true.
500   continue

c     B A C K   S U B S T I T U T I O N
      do 1000 ii = myid,N,Nprocs
      i = N+1-ii
      do 900 jj = 0, i-1
               j = N - jj
               when( done(j,2) ) continue
               do 800 k = 1,4
               x(row(i),k) = x(row(i),k) - a(row(i),j)*x(row(j),k)
800            continue
900   continue
               do 920 k = 1,4
               x(row(i),k) = x(row(i),k) /a(row(i),i)
920            continue
      done(i,2) = .true.

      do 950 j = 1,N
      do 940 k = 1,4
         resid(row(j),k) = resid(row(j),k) - a(row(j),i)*x(row(i),k)
940   continue
950   continue
1000  continue

c     Use semaphore ISYNCH to signal fellow processors that your work is done, so far.
      call CFlock(ICFret,1,'isynch')
      isynch = isynch + 1
      call CFulck(ICFret,1,'isynch')

      do 1100 k = 1,4
         temp(k) = 0.0
1100  continue

c     Wait on fellows to get done with their work and then compute residual:
      when ( isynch .ge. Nprocs ) continue
      do 1500 i = myid,N,Nprocs
      do 1400 k = 1,4
               temp(k) = temp(k) + resid(i,k)**2
1400  continue
1500  continue

      call CFlock(ICFret,1,'rnorm')
               do 2000 k = 1,4
               rnorm(k) = rnorm(k) + temp(k)
2000           continue
      call CFulck(ICFret,1,'rnorm')

      return
      end
```

APPENDIX TWO: MEASURED PERFORMANCE DATA

We present in tabular form the data sets plotted in Section 4. The entries are clock ticks (= to 1/50 of a second) and, for more than 1 processor, the resulting efficiency.

Number of Integration Points

Processors	100	500	1,000	5,000	10,000	25,000	32,000
1	2	9	16	79	158	394	473
2	3 .33	6 .75	10 .80	42 .94	81 .98	199 .99	238 .99
3	4 .17	6 .50	9 .59	31 .85	56 .94	135 .97	161 .98
4	5 .10	6 .38	10 .40	25 .79	45 .88	103 .96	124 .95
5	5 .08	6 .30	8 .40	21 .75	36 .88	86 .92	101 .94
rel error	$2 \cdot 10^{-4}$	$8 \cdot 10^{-6}$	$2 \cdot 10^{-6}$	$2 \cdot 10^{-7}$	$4 \cdot 10^{-7}$	$8 \cdot 10^{-7}$	$3 \cdot 10^{-7}$

Problem 1: Trapezoidal Rule for $f(x) = e^x$.

Number of Indices

Processors	10	25	40	55	70	80	85
1	2	11	25	48	77	101	114
2	4 .25	8 .69	15 .83	26 .92	41 .94	53 .95	60 .95
3	4 .17	7 .52	12 .69	20 .80	30 .86	37 .91	42 .90
4	5 .10	8 .34	12 .52	18 .67	24 .80	31 .81	33 .86
5	4 .10	6 .37	9 .56	13 .74	20 .77	24 .84	27 .84

Problem 2: Compute sum of products of expression.

Array Size (square)

Processors	50	100	150	200	250	300	400	500
1	8	28	64	112	175	251	446	697
2	6	16	34	58	89	127	224	349
	.67	.88	.94	.97	.98	.99	1.0	1.0
3	7	13	25	41	63	87	153	235
	.38	.72	.85	.91	.93	.96	.97	.99
4	7	13	21	33	49	68	117	179
	.29	.54	.76	.85	.89	.92	.95	.97
5	6	13	17	29	39	56	96	145
	.27	.43	.75	.77	.90	.90	.93	.96

Problem 3: Compute sum of products of array elements.

Number of vector elements

Processors	2,500	10,000	25,000	50,000	75,000	100,000	150,000	200,000
1	6	20	51	102	153	204	304	404
2	5 .60	12 .83	27 .94	54 .94	79 .97	105 .97	156 .97	208 .97
3	5 .40	11 .61	21 .81	38 .89	55 .93	72 .94	106 .96	141 .96
4	6 .25	11 .45	19 .67	31 .82	43 .89	56 .91	82 .93	108 .94
5	5 .24	8 .50	17 .60	28 .73	37 .83	45 .91	69 .88	89 .91

Problem 4: Compute sum of reciprocols of non-zero elements.

Number of Students and Grades

Processors	25×25	50×50	100×100	150×150	200×200	225×225	250×250	300×300
1	4	12	48	107	188	238	293	422
2	6 .33	11 .55	28 .86	58 .92	99 .95	124 .96	151 .97	215 .98
3	7 .17	12 .33	24 .67	43 .83	70 .90	87 .91	106 .92	148 .95
4	8 .13	12 .25	20 .60	36 .74	57 .83	69 .86	83 .88	114 .93
5	9 .09	11 .22	18 .53	35 .61	48 .78	58 .82	70 .84	96 .88

Problem 5: Grading Program – Pessimistic Version: Times include creating processes.

Number of Students and Grades

Processors	25×25	50×50	100×100	150×150	200×200	225×225	250×250	300×300
1	4	12	47	106	186	236	291	416
2	5 .4	9 .67	27 .87	55 .96	96 .97	122 .97	148 .98	212 .98
3	3 .44	8 .5	20 .78	41 .86	68 .91	84 .94	103 .94	144 .96
4	4 .25	6 .5	18 .65	33 .80	51 .91	62 .95	80 .91	111 .94
5	3 .27	5 .48	15 .63	26 .82	40 .93	50 .94	61 .95	89 .94

Problem 5: Grading Program – Optimistic Version: Does not include creating processes.

Matrix Order

Processors	50	100	250	500	1,000	1,500	2,000	2,500	5,000	10,000
1	12	21	55	113	241	356	516	642	1384	2975
2	11	17	36	67	134	194	280	345	735	1571
	.55	.62	.76	.84	.90	.92	.92	.93	.94	.95
3	12	16	31	52	96	137	193	238	496	1056
	.33	.44	.59	.72	.84	.87	.89	.90	.93	.94
4	15	18	31	47	82	112	153	187	384	805
	.20	.29	.44	.60	.73	.79	.84	.86	.90	.92
5	19	20	32	46	73	97	133	159	318	655
	.13	.21	.34	.49	.66	.73	.78	.81	.87	.91

Problem 6: Tridiagonal Matrix Solver

Number of evaluation points

Processors	25	50	100	1,000	2,000	4,000	10,000
1	14	47	116	134	241	435	975
2	9 .78	27 .87	60 .97	73 .92	127 .95	226 .96	513 .95
3	9 .52	20 .78	44 .88	52 .86	88 .91	157 .92	348 .93
4	9 .39	19 .62	35 .83	44 .76	73 .83	126 .86	271 .90
5	10 .28	17 .55	30 .77	36 .74	60 .81	104 .84	229 .85

Problem 7: LaGrange polynomial interpolation.

Number of Data Points

Processors	10	25	50	70	100	150	200
1	9	57	233	459	940	2122	3778
2	10 .45	53 .54	219 .53	404 .57	863 .54	1933 .55	3732 .51
3	11 .27	58 .33	210 .37	402 .38	816 .38	1831 .39	3225 .39
4	15 .15	58 .25	197 .30	401 .29	817 .29	1746 .30	3322 .28
5	17 .11	68 .17	217 .21	399 .23	823 .23	1784 .24	3053 .25

Problem 8: Divided difference table - tightly synchronized version.

Number of Data Points

Processors	10	25	50	100	150	200	250	300
1	2	6	15	51	103	175	265	371
2	4	8	20	56	110	181	276	383
	.25	.38	.38	.46	.47	.48	.48	.48
3	5	11	21	59	115	189	281	393
	.13	.18	.24	.29	.30	.31	.31	.31
4	4	12	23	62	119	191	291	404
	.12	.12	.16	.21	.22	.23	.23	.23
5	5	11	25	64	121	202	298	416
	.08	.11	.12	.16	.17	.17	.18	.18

Problem 8: Divided difference table - naive version which results in almost sequential execution. Note that asympotic efficiency for N processors is approximately $1/N$.

Size of 2D Array (square)

Processors	10	25	40	50	60	75	90	100
1	15	86	223	352	512	807	1169	1449
2	13	49	116	181	261	414	589	729
	.58	.88	.96	.97	.98	.97	.99	.99
3	14	38	85	126	184	285	407	496
	.36	.75	.87	.93	.93	.94	.96	.97
4	15	34	70	99	143	221	303	381
	.25	.63	.80	.88	.90	.91	.96	.95
5	19	34	63	88	119	180	254	310
	.16	.51	.71	.80	.86	.90	.92	.93

Problem 9: Averaging neighbors in 2D array.
10 Passes.

Matrix Dimension

Processors	15	25	35	40	45	55	65	80	100
1	56	158	327	441	577	919	1369	2269	3961
2	38 .74	95 .83	186 .88	246 .90	317 .91	495 .93	726 .94	1188 .95	2049 .97
3	34 .55	79 .67	144 .76	186 .79	236 .81	360 .85	519 .88	836 .90	1429 .92
4	34 .41	71 .56	127 .64	158 .70	200 .72	303 .76	425 .81	672 .84	1124 .88
5	32 .35	73 .43	117 .56	148 .60	175 .66	271 .68	367 .75	567 .80	950 .83

Problem 10: Gaussian elimination with partial pivoting.

# of Vector Elements

Processors	250	500	1,000	2,500	7,500	12,500	25,000	50,000
1	23	33	51	108	298	488	961	1907
2	19 .61	25 .66	34 .75	63 .86	159 .94	252 .97	489 .98	963 .99
3	20 .38	23 .48	29 .59	49 .73	112 .89	174 .93	333 .96	649 .98
4	20 .29	23 .36	28 .46	42 .64	90 .83	137 .89	256 .94	492 .97
5	21 .22	23 .29	27 .38	38 .57	77 .77	115 .85	209 .92	381 1.0

Problem 11: Data filtering.

Size of ABIG

Processors	10×10	25×25	50×50	100×100	150×150	200×200	250×250	300×300
1	1	2	8	33	74	131	205	296
2	2 .25	3 .33	7 .57	18 .92	39 .95	69 .95	106 .97	151 .98
3	2 .17	4 .17	6 .44	15 .73	28 .88	48 .91	72 .95	102 .97
4	3 .08	5 .10	8 .25	14 .59	24 .77	37 .89	57 .90	80 .93
5	4 .05	5 .08	7 .23	12 .55	22 .67	33 .79	45 .91	67 .88

Problem 12: Construction of a big array.

Number of Vector Elements

Processors	50	100	250	500	1,000	2,500	5,000	7,500	10,000
1	4	8	17	35	69	172	343	513	683
2	24 .50	6 .67	11 .77	20 .88	37 .93	88 .98	174 .99	259 .99	343 1.0
3	5 .27	6 .44	9 .63	16 .73	27 .85	61 .94	118 .97	175 .98	231 .99
4	5 .20	5 .40	10 .43	12 .73	21 .82	47 .91	92 .93	134 .96	177 .96
5	5 .16	5 .32	11 .31	11 .64	18 .77	41 .84	76 .90	110 .93	144 .95

Problem 13: Transform a vector, sum squares of elements.

Number of Evaluation Points

Processors	10	25	50	100	250	500	1,000	2,500	3,300
1	2	5	7	15	37	74	148	369	488
2	3 .33	4 .62	7 .5	10 .75	21 .88	39 .95	77 .96	187 .99	247 .99
3	4 .17	5 .33	6 .39	9 .56	17 .73	29 .85	53 .93	128 .96	166 .98
4	4 .13	6 .21	8 .22	9 .42	15 .62	22 .84	41 .90	98 .94	128 .95
5	5 .08	5 .20	6 .23	11 .27	14 .53	22 .67	38 .78	81 .91	106 .92

Problem 14: Test 4 integrators on 10 functions.

Number of Evaluation Nodes

Processors	50	100	250	500	1,000	3,500	5,000	10,000
1	10	17	38	75	148	508	726	1447
2	8 .63	12 .71	23 .83	40 .94	76 .97	258 .98	365 .99	727 .99
3	9 .37	11 .52	18 .70	30 .83	55 .90	174 .97	248 .97	487 .99
4	9 .28	11 .39	17 .56	26 .72	44 .84	134 .95	188 .97	370 .98
5	11 .18	12 .28	16 .48	24 .63	38 .78	110 .92	154 .94	298 .97

Problem 15: Comparison of interpolation methods.

Size of Matrix

Processors	5	10	15	20	25	30	33	35	40
1	15	46	101	178	279	407	496	563	749
2	17 .44	38 .61	74 .68	122 .73	182 .77	262 .78	316 .78	359 .78	454 .82
3	19 .26	42 .37	68 .50	107 .55	162 .57	220 .62	262 .63	290 .65	378 .66
4	25 .15	48 .24	74 .34	106 .42	154 .45	217 .47	243 .51	271 .52	346 .54
5	28 .11	48 .19	74 .27	112 .32	164 .34	202 .40	241 .41	270 .42	339 .44

Problem 16: Solve Hilbert problem with multiple right sides.